KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Compile-Time Garbage Collection for the Declarative Language Mercury

Promotoren :
Prof. Dr. ir. M. BRUYNOOGHE
Prof. Dr. ir. G. JANSSENS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Nancy Mazur

May 2004

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

Compile-Time Garbage Collection for the Declarative Language Mercury

Jury :
Prof. Dr. ir. J. Berlamont, voorzitter
Prof. Dr. ir. M. Bruynooghe, promotor
Prof. Dr. ir. G. Janssens, promotor
Prof. Dr. B. Demoen
Prof. Dr. ir. E. Steegmans
Prof. Dr. ir. A. King
(University of Kent, United Kingdom)
Prof. Dr. ir. Z. Somogyi
(University of Melbourne, Australia)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Nancy Mazur

U.D.C. 681.3*D3

May 2004

D/2004/7515/45
ISBN 90–5682–505–4

Abstract

One of the key advantages of modern programming languages is that they free the programmer from the burden of explicit memory management. Usually, this means that memory management is delegated to the run-time system by the use of a *run-time garbage collector* (RTGC). Basically, a RTGC is a dedicated process that is run in parallel with the user program. Whenever the user program needs to store some data, the RTGC provides the desired memory space. At regular intervals, the RTGC reviews the uses of the allocated memory space, and recovers those memory cells that have become *garbage*, *i.e.* , that can not be accessed any more by the user program.

A complementary form of automatic memory management is *compile-time memory management* (CTGC), where the decisions for memory management are taken at compile-time instead of at run-time. The compiler determines the lifetime of the variables that are created during the execution of the program, and thus also the memory that will be associated with these variables. Whenever the compiler can guarantee that a variable, or more precisely, parts of the memory resources that this variable points to at run-time, will never ever be accessed beyond a certain program instruction, then the compiler can add instructions to deallocate these resources at that particular instruction without compromising the correctness of the resulting code. If the program instruction is followed by a series of instructions that require the allocation of new memory cells, then the compiler can replace the sequence of deallocation and allocation instructions, by instructions updating the garbage cells, hence *reusing* these cells.

We study the technique of compile-time garbage collection in the context of Mercury, a pure declarative language. A key element of declarative languages is that they disallow explicit memory updates (which are common operations in most other programming paradigms) but they rely instead on term construction and deconstruction to manipulate the program data. This places a high demand on the memory management and makes declarative languages a primary target for compile-time garbage collection. Moreover, the clear mathematical foundations of Mercury, being a pure declarative language, makes the development of the program analyses that are necessary for CTGC feasible.

In this thesis we define a number of semantics for the logic programming language Mercury and formally establish the equivalence between them; we use these semantics to formalise the different program analysis steps that are needed to implement a basic CTGC system for Mercury and prove their safeness. We extend this basic CTGC system such that it is able to correctly deal with programs organised into modules and we implement a complete CTGC system within the Melbourne Mercury Compiler. To the best of our knowledge, this is the first and only complete CTGC system that has ever been built for a programming language.

Acknowledgements

As a little girl I was telling everybody I'd be "Dr. Ir. Nancy Mazur" one day, without having the faintest idea what this meant. Today, a number of years later, I am proud to say that I have reached this goal, and I do think I can say that I know what it means now.

I would like to thank my supervisors, Professor Maurice Bruynooghe and Professor Gerda Janssens for having given me the opportunity to work on this project, for their guidance and confidence, for I surely was not an easy pupil. I thank the members of my jury, Professor Bart Demoen, Professor Eric Steegmans, Professor Andy King and Professor Zoltan Somogyi, for proofreading my work and providing me with detailed and valuable comments.

Special thanks go to Professor Maurice Bruynooghe and Professor Zoltan Somogyi for making my visit to the University of Melbourne possible. I also thank the members of the Mercury Team for helping me get started with the Melbourne Mercury compiler. I would like to thank Peter Ross in particular. Without him, the implementation of my system within the compiler would not have been possible.

Over the years I have been involved with the graduate course "Methodiek van de Informatica". The contact with the students and the interesting evolution of this course made this a valuable and fascinating experience for me. I would like to thank Professor De Vlaminck and my MI-colleagues for sharing their enthusiasm for this course with me.

I would like to thank the many colleagues at the department making it such an interesting place to work. Special thanks go to Denise Brams, Karin Michiels, Margot Peeters, Esther Renson and Lieve Swinnen for their daily assistance in all these small yet necessary administrative tasks in which I'd be hopelessly lost. Thanks also to our system administrators for making it all work.

Furthermore, I would like to thank Marc Denecker and Jacques Riche. It was a pleasure sharing an office with you during my first year at the department, and I still remember our fascinating yet sometimes endless ;-) discussions about Logic Programming and The World in general. Dear Wim Vanhoof and Sofie Verbaeten, I think we can proudly say that our little L.00.10 office was the best

office ever. You made it happen. Thanks! Wim, our working trip to Australia wouldn't have been that fantastic if not with a travel mate like you. Moreover, even at your busiest moments, you'd always have time for discussion and extra encouragements. For all that and more, thank you.

Beloved friends, I know that in days of stress I can always count on you. Jo and Barbara, Joris and Isobel, Raf and Lucie: you are great cooks! And I hope we can continue our tradition for many other years. Wim, Nico and Siska, Dieter, Stefan, Veerle (and Bandy), Raf, Luk and Hilary, Bart and Sandra, you adopted me in your group and rewarded me with a great feeling of friendship. Thank you. Polien, Patricia, Natasja, Sofie, you're great young moms, and it is always fascinating to discuss all possible baby-kids-issues with you.

Dear Kurt and our little Hannah, thank you for your love and incredible patience. *Davidek i Kristien, kochana oma, dzięki. Ela i Stasio, Mama i Papa, wiem że was nie mogłam dziękowac, ale znacze piosenkę "*Róbmy swoje*", i dla tego: dzięki, tak sobie.*

Nancy Mazur
May 2004

Contents

Nomenclature

Domains

$\mathcal{AL}$	Domain of abstract liveness descriptions where $\mathcal{AL}$ is equivalent to $\langle \wp(\overline{\mathcal{SD}}_{VI}), \wp(\overline{\mathcal{D}}_{VI})\rangle$, page 180
$\langle \mathcal{AL}, \sqsubseteq_{al}\rangle$	Ordering in $\mathcal{AL}$, page 180
$\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \bot_X, \top_X\rangle$	Complete lattice X, page 14
$\langle X, \subseteq_X, \cup_X, \cap_X, \bot_X, \top_X\rangle$	Complete lattice X, page 14
$\mathcal{CL}$	Concrete domain of collecting liveness descriptions, *i.e.*, $\mathcal{CL} = \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle)$, page 173
$\langle \mathcal{CL}, \sqsubseteq_{cl}\rangle$	Ordering in $\mathcal{CL}$, page 174
$\overline{\mathcal{D}}_{VI}$	Set of abstract data structures of the variables in VI, page 129
$\mathcal{D}_{VI}$	Set of context-free data structures of the variables in VI, page 115
$\overline{\mathcal{D}}_X$	Set of abstract data structures of X, page 129
$\mathcal{D}_X$	Set of context-free data structures of X, page 115
$\langle Eqn^+, \mathcal{D}_{VI}\rangle$	Concrete domain of data structures, page 115
$\langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle$	Concrete domain of liveness descriptions, page 173
$\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$	Set of sharing sets over the variables VI, page 116
$\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle$	Concrete domain of data structure sets, page 163
Eqn^+	Set of solvable existentially quantified ex-equations, page 16
Eqn	Set of existentially quantified ex-equations, page 16
$\langle Eqn^+, \models\rangle$	Ordering of existentially quantified ex-equations, page 16
Σ	Finite set of function symbols, page 14
$\wp(\overline{\mathcal{D}}_{VI})$	Domain of abstract data structure sets, page 166
$\langle \wp(\overline{\mathcal{D}}_{VI}), \sqsubseteq_{ad}\rangle$	Ordering in $\wp(\overline{\mathcal{D}}_{VI})$, page 167
$\langle \wp(\mathcal{D}_{VI}), \subseteq_{cd}\rangle$	Ordering of (concrete) context-free data structure sets, page 164
$\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$	Concrete domain of collecting data structure sets, page 163
$\langle \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle), \sqsubseteq_{cd}\rangle$	Ordering of (concrete) collecting data structure sets, page 164
$\langle \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle), \sqsubseteq_c\rangle$	Ordering in $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, page 120
$\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$	Set of collecting sharing sets over VI, page 117
Pos	The domain of positive boolean equations, page 73

$Pos_\perp$	Domain *Pos* extended with false, page 73
Π	Finite set of predicate symbols, page 14
$\wp(\overline{\mathcal{SD}}_{VI})$	Set of sets of abstract sharing pairs over VI, page 129
$\langle \wp(\overline{\mathcal{SD}}_{VI}), \sqsubseteq_a \rangle$	Ordering in $\wp(\overline{\mathcal{SD}}_{VI})$, page 130
$\langle \wp(\mathcal{SD}_{VI}), \subseteq_c \rangle$	Ordering in $\wp(\mathcal{SD}_{VI})$, page 119
$\langle \mathcal{RI}, \ll_r \langle$	Ordering in $\mathcal{RI}$, page 230
$\mathcal{RI}$	Reuse information domain where $\mathcal{RI}$ is the shorthand notation for $\wp(\langle\!\langle \wp(\overline{\mathcal{D}}_{VI}), \wp(\overline{\mathcal{D}}_{VI}), \wp(\overline{\mathcal{SD}}_{VI}) \rangle\!\rangle)$, page 229
$\overline{\mathcal{SD}}_{VI}$	Set of abstract structure sharing pairs over variables in VI, page 129
$\mathcal{SD}_{VI}$	Set of context-free sharing data structures of the variables VI, page 115
Selector	Domain of term/type selectors, *i.e.*, sequences in $\Sigma \times \mathbb{N}$, page 108
TSelector	Domain of t-selectors, page 253
$\Sigma_\mathcal{T}$	Finite set of type constructors, page 23
$\mathcal{T}(\mathcal{V}, \Sigma)$	Set of terms, page 15
$\mathcal{T}(\Sigma_\mathcal{T}, \mathcal{V}_\mathcal{T})$	Set of types, page 23
$\mathcal{V}_\mathcal{T}$	Finite set of type variables, page 23
$\mathcal{V}$	Finite set of variables of a logic program, page 14
VI	Variables of interest, page 115

Functions, Operations

altclos	Alternating closure operation, page 134
$\mathsf{altclos}_i$	Alternating closure limited to alternating paths of length i, page 217
$\mathsf{altclos}_{>i}$	Alternating closure limited to alternating paths of length $> i$, page 217
$\mathsf{altclos}_\rightarrow$, $\mathsf{altclos}_\leftarrow$	Directional alternating closure operations, page 217
init_c, comb_c, add_c	Auxiliary functions in $\mathcal{S}em_M(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle))$, page 120
$\mathtt{det}(S)$	Determinism of a syntactic entity S, page 45
$\gamma^\mathcal{D}$	Concretisation function mapping abstract data structures onto sets of concrete data structures, page 168
$\gamma^\mathcal{L}$	Concretisation function between $\mathcal{AL}$ and $\mathcal{CL}$, page 181
$\gamma^\mathcal{S}$	Concretisation function between the structure sharing domains, page 130
$\mathtt{in}(S)$	Input variables to an expression S, page 45
init_a, comb_a, add_a	Auxiliary functions in $\mathcal{S}em_M(\wp(\overline{\mathcal{SD}}_{VI}))$, page 135
$\mathsf{live}_a(i, \overline{A}, \overline{L}_0)$	Function for the abstract liveness at a program point (i) with abstract structure sharing $\overline{A}$ and abstract global liveness $\overline{L}_0$, page 180
$\mathsf{live}_a(i, \overline{A}_g, \overline{A}_l, \overline{L}_0)$	Improved live_a-function, page 185

$\mathsf{live}(i, \langle e, C\rangle, L_0)$	Function for the concrete liveness at a program point (i) with structure sharing description $\langle e, C\rangle$ and global liveness L_0, page 171
$\mathsf{live}(i, \langle e_0, C_0\rangle, \langle e_{l,i}, C_{l,i}\rangle, L_0)$	Improved live-function, page 185
$\mathtt{out}(S)$	Output variables to an expression S, page 45
$\mathtt{paths}(p)$	Ordered sequence of execution paths of a procedure p, page 42
$\overrightarrow{p_i}$	The i'th execution path of a procedure p, page 42
$\mathtt{pp}$	Program points, e.g. $pp(l)$ – program point of a literal l, $\mathtt{pp}(g)$ – program points within a goal g, $\mathtt{pp}(p)$ – program points within a procedure p, page 42
$\mathtt{pre}(g), \mathtt{post}(g)$	Preceding/Following program points of goal (g), page 43
$\mathtt{pre}(i), \mathtt{post}(i)$	Preceding/Following program points of point (i), page 43
$(e)\|_V$	Restriction of an entity e to the variables in V, page 18
$\overset{\mathsf{comb}^{\mathcal{A}}}{\longrightarrow}$	Relative pseudo-complement generalised w.r.t. the exact combination operation $\mathsf{comb}^{\mathcal{A}}$, e.g. $\delta_a \overset{\mathsf{comb}^{\mathcal{A}}}{\longrightarrow} \delta_b$, page 299
ρ	Renaming substitution, page 15
$\rho_{X\to Y}(E)$	Renaming of E w.r.t. two sets of variables, page 18
$\rho_{\overline{V_1}\to\overline{V_2}}(E)$	Renaming of E w.r.t. two sequences of variables, page 18
$AL\sharp R_i$	The liveness information described by AL meets the reuse information described by R_i, c.f. Equation (10.3), page 215
$\bullet$	Concatenation of selectors, e.g. $s_1 \bullet s_2$, page 108
$\lhd_{\mathsf{t}}$	T-selector covering, e.g. $s \lhd_{\mathsf{t}} s^{\sharp}$, page 254
$\leftrightsquigarrow_{\mathsf{t}}$	Mapping of a t-selector, page 254
$\mathcal{T}_\tau$	Term tree of a term τ, page 109
$\mathtt{type}(X, p), \mathtt{type}(X)$	Type of a variable X (in procedure p), page 44
$\mathcal{TG}_{\mathsf{t}}$	Type graph of a type t, page 113
$\mathcal{TT}_{\mathsf{t}}$	Type tree of a type t, page 108
$T[k, e]$	Update a table T with the tuple $\langle k, e\rangle$, page 59
$Var(a)$	The variable appearing in a – only applicable to a if a is known to relate to exactly one variable, page 15
$Vars(a)$	The set of variables appearing in the object a, page 15

Semantics

$\mathcal{S}em_{M\delta}$	Differential semantics for Mercury, Fig. 5.6, page 81
$\mathcal{S}em_{M\delta}(\wp(Eqn^+))$	Concrete differential semantics for Mercury, page 88
$\mathcal{S}em_{M\delta^+}$	Augmented Differential Semantics, page 185
$\mathcal{S}em_{M\delta^+}(\mathcal{AL})$	Differential Abstract Liveness Derivation, page 186
$\mathcal{S}em_{M\delta^+}(\mathcal{CL})$	Differential Concrete Liveness Derivation, page 186
$\mathcal{S}em_{M\bullet}$	Goal-dependent part based on the goal-independent based semantics $\mathcal{S}em_{M\star}$, Fig. 5.10-5.11, page 93

$\mathcal{S}em_{M\star}$	Goal-independent semantics of a Mercury rulebase, Fig. 5.8-5.9, page 92
$\mathcal{S}em_{M\bullet^+}$	Augmented goal-independent based semantics (for liveness analysis), page 209
$\mathcal{S}em_{M\star p}$	Goal-independent semantics with pre-annotations, Fig. 5.12-5.13, page 99
$\mathcal{S}em_{M\bullet p}$	Goal-dependent part based on the goal-independent based semantics $\mathcal{S}em_{M\star p}$ with pre-annotations, Fig. 5.14-5.15, page 99
$\mathcal{S}em_{M}$	Natural semantics for Mercury, Fig. 5.4, page 78
$\mathcal{S}em_{M}(\wp(Eqn^+))$	Concrete goal-dependent semantics for Mercury, page 78
$\mathcal{S}em_{M^+}$	Augmented Natural Semantics Section 8.3.2, page 174
$\mathcal{S}em_{M^+}(\mathcal{AL})$	Abstract Liveness Derivation, page 181
$\mathcal{S}em_{M^+}(\mathcal{CL})$	Concrete Liveness Derivation, page 175
$\mathcal{S}em_{S}$	Natural semantics for Simple Mercury(Fig. 5.1-5.3) , page 65
$\mathcal{S}em_{S}(\wp(Eqn^+))$	Concrete natural semantics for Simple Mercury, page 67
$\mathcal{S}em_{\Omega}$	Increased precision optimisation derivation framework, page 306
$\mathcal{S}em_{\omega}$	Optimisation derivation framework for Mercury, page 301
$\mathcal{S}em_{\omega+}$	Variation on $\mathcal{S}em_{\omega}$, page 303

Symbols

$\tau, \tau_1, \ldots$	Terms, page 15
$EC, EC_1, EC_2, \ldots$	Elements in $\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$, page 117
$ECS, ECS_1, ECS_2, \ldots$	Elements in $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, page 117
f, g, h	Function symbols, page 14
p, q, r	Predicates/Procedures, page 14
$X, Y, First, \ldots$	Variables, page 14
$\langle \overline{A}, \overline{L}\rangle$	Decomposed abstract liveness description, page 180
$AL, AL_1, AL_2, \ldots$	Abstract liveness descriptions, page 180
$\propto$	Safe approximation, e.g. $y \propto x$, meaning that y safely approximates x, page 53
$X^{\overline{sx}}$	Abstract data structure, page 128
$(X - Y)$	Abstract sharing pair, page 129
$\langle e, X^{sx}\rangle$	Concrete data structure, page 115
$ED, ED_1, \langle e, D\rangle, \ldots$	Concrete data structure set, page 163
$EDS, EDS_1, EDS_2, \ldots$	Concrete collecting data structure sets, page 163
$\langle e, C, L\rangle$	Liveness description with environment e, structure sharing set C, and current liveness component L, page 173
$CL, CL_1, CL_2, \ldots$	Collecting liveness descriptions, page 173
δ_i^m	Call requirement at a program point i, page 299
$\langle\!\langle D_i, U_i, A_i\rangle\!\rangle$	Reuse information tuple, page 215
μ	Minimal requirement, page 298

$\wp(S)$	Powerset of a set S, page 13
$\overline{R}, \overline{R_1}$	Compacted reuse information tuples, page 229
$R, R_1, R_2, \ldots$	Reuse information tuples, page 229
$RI, RI_1, RI_2, \ldots$	Collecting reuse information tuples, *i.e.*, $RI \in \mathcal{RI}, RI_1 \in \mathcal{RI}, \ldots$, page 229
$\bowtie$	Compatible selectors, e.g. $s_1 \bowtie s_2$, page 110
$\boldsymbol{s}^\sharp, \boldsymbol{s_1}^\sharp, \ldots$	T-selectors, page 253
$\overline{a}$ or $a_1, \ldots, a_n$	Sequence of elements, page 13
$\langle e, (X^{s_X} - Y^{s_Y}) \rangle$	Concrete sharing pair, page 115
θ, σ	Variable substitutions, page 15

Chapter 1

Introduction

1.1 Memory Management

One of the key advantages of modern programming languages is that they free the programmer from the burden of explicit memory management. Usually, this means that memory management is delegated to the run-time system by the use of a so called *run-time garbage collector* (RTGC). Basically, a run-time garbage collector can be seen as a dedicated process that is run in parallel with the user program. Whenever the user program needs to store some information, the RTGC is asked to provide the desired memory space. At regular intervals, or if the memory space tends to become full, the RTGC analyses all the data that is stored in memory: data that is still *reachable* by the user program is called *live*, while data that can definitely not be accessed anymore by the user program is classified as *dead*. While live data must carefully be kept, all dead data may safely be deallocated, and the corresponding memory can be made available for subsequent allocations.

While this scheme of automatic memory management has reached the big public mainly by the popular programming language Java and its run-time environment (Gosling and McGilton 1995), the history of run-time garbage collectors dates back to LISP, and the development of the so called *declarative programming languages* in general. Indeed, in the latter family of programming languages, even simple destructive updates, which are common operations in the imperative programming paradigm, are prohibited. In this context, updating a particular field within a structure saved in memory means to create a copy of that structure and update that specific field. This is one of the cornerstones of their declarativeness. However, updating data this way is time consuming, and more importantly, leads to large memory consumption, making the need for good memory management even more important.

While the run-time system can take care of automatic memory management, it also has its disadvantages:

- Obviously, a RTGC has an associated cost with respect to execution time as well as memory space. In extreme cases, such as programs developed for embedded systems, this form of memory management may have to be slimmed down or even completely removed.

- The run-time system may not have all the information about the program at its disposition, and to guarantee the safeness of the deallocations, may have to overestimate the live data, leading to a larger memory footprint than (what could have been) expected. Garbage collectors having to act in such uncooperative environments are called *conservative* garbage collectors.

- The time between the moment that a particular structure becomes dead, and the moment where it is deallocated by the garbage collector can be substantial. This may unnecessarily clutter the memory usage of the program.

- The lack of destructive update in declarative languages makes that — in order to change a value in a data structure — the structure needs to be deconstructed, and immediately thereafter be reconstructed with only a few values changed. Rather than deallocating the memory occupied by the former structure and allocating new memory for the latter structure, the same block of memory could be (re)used.

As a consequence, even in the presence of garbage collectors, declarative programmers were taught a bag of tricks allowing to circumvent the lack of destructive updates. An example is the use of open ended data structures such as difference lists which are often used in Prolog for the purpose of a better memory behaviour. Moreover, declarative languages often include primitive operations or language constructs allowing some form of direct memory management. A typical example is the use of assert and retract predicates in Prolog. These predicates do indeed allow the destructive update of memory blocks, yet these destructive updates are implemented as *side-effects*, hence, do not preserve the clean declarative semantics of the program: performing the same query or function twice may not lead to exactly the same answer. In more recent declarative languages, in which purity is kept as an essential characteristic of the language, other, more sophisticated techniques have been developed, the most popular being the use of so called unique objects (Somogyi, Henderson, and Conway 1996; Bekkers and Tarau 1995; Wadler 1992), the uniqueness of which are automatically verified by the compiler.

Whether pure or not pure, sophisticated or not sophisticated, all these *ad hoc* techniques and approaches may have the advantage of saving some memory usage, yet their use is by definition cumbersome as it does not fit the declarative

paradigm where the programmer should not have to worry about memory management in the first place. This principle has continued to be the driving force of a large community of researchers to further investigate the potential of run-time garbage collection, yet it has also led to the interesting and challenging research area of *compile-time garbage collection*.

The principle of compile-time garbage collection (CTGC) is to determine at *compile-time* when memory blocks become dead during the execution of the program. The process is meant to be completely automatic, and relies on program analysis for determining the dead structures. The goal of a CTGC system is to downsize the memory usage of a program, reduce the responsibility of the run-time collector, hence also hopefully its overhead, and perhaps as an end-effect, reduce the overall execution time of the program.

The dream of compile-time garbage collection has existed since years, yet it is only recently, with the maturity of program analysis in general, that this dream could be realised into a complete working system. In this work we realise this dream in the particular context of the modern logic programming language Mercury.

1.2 An intuitive example

We give a brief sketch of how compile-time garbage collection provides for automatic memory management by the use of a small example. This allows us to highlight the issues and challenges involved with this technique.

We assume some familiarity with logic programming, although similar situations can be sketched in other programming languages too.

Consider a predicate that updates the salary of an employee in a database. If we assume that an employee is simply represented as a term with a list of arguments consisting of their name, birthday, and salary, then the predicate may be written as:

```
updateSalary(EmployeeRecord, NewSalary, NewRecord) :-
    EmployeeRecord = employee(Name, Birthday, OldSalary),
    NewRecord = employee(Name, Birthday, NewSalary).
```

Note that indeed, as we mentioned earlier, a simple update in a pure logic programming language consists of creating a new term that serves as a copy of the original term, with the only difference that one of its fields will have a different value.

Now assume that the above predicate is part of a larger administrative software package, and that at some moment during the execution of that package, a variable JackRecord is bound to the structured term:

```
employee("Jack␣Newman", 19490319, 40000)
```

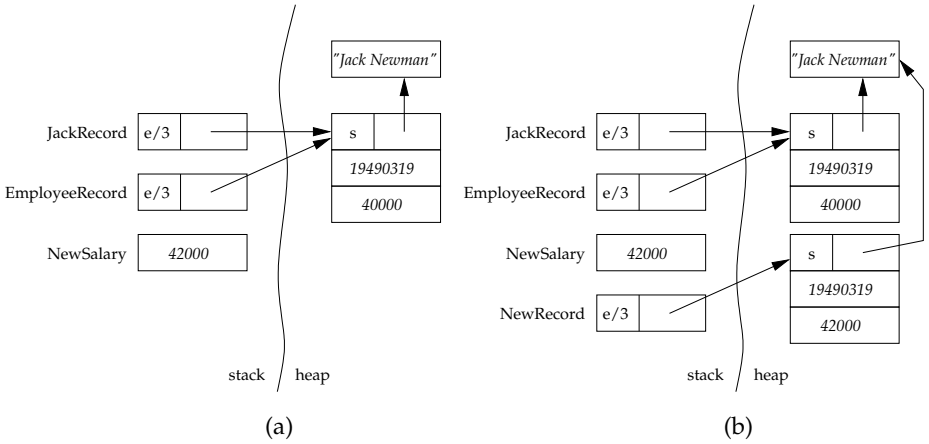


Figure 1.1: Sketch of the call updateSalary(JackRecord, 42000, NewJackRecord), with JackRecord bound to employee("Jack␣Newman", 19490319, 40000). Parts (a) and (b) picture the memory layout *before*, resp. *after* the body of the predicate is considered.

representing the record of an employee named Jack Newman, born on the 19th of March in 1949, and currently receiving 40000$ a year. Updating his wages can be achieved by calling the above defined predicate, e.g.,

```
updateSalary(JackRecord, 42000, NewJackRecord)
```

Figure 1.1 sketches the memory layout of that call, *before* executing the body of the predicate (a), and right *after* having executed them (b).

Now suppose that variable JackRecord was the only reference to the old situation of Jack's record within the program, then the memory fields representing that record can be considered as *dead* which a run-time garbage collector can deallocate in one of its future cycles. This is depicted in Figure 1.2 (a), where the dead data is marked using grey boxes with dashed lines. This is how a run-time garbage collection would behave.

Now what if we were able to detect at *compile-time* that for some calls of updateSalary the term associated to the first argument may become dead? Then we could compile the program to include the low-level instructions which deallocate these dead memory cells, or we could push program analysis even further and try to find possibilities of reusing these dead cells for constructing the new employee record, hence, producing the low-level instructions to perform the destructive update of the initial record. For the run-time process this means that the deallocation followed by the allocation is replaced by instructions for altering one simple field, namely the field of the salary of the employee, and copying

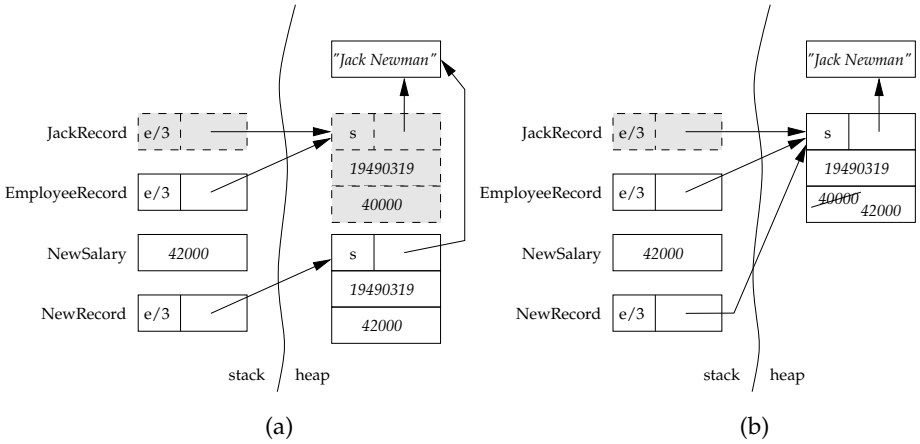


Figure 1.2: (a) After the call, the memory cells of EmployeeRecord become garbage (shown as grey boxes). (b) Instead of generating garbage, the dead cells can be reused for creating the new structure bound to NewRecord.

the pointer to the original term to the new variable NewRecord. This situation is depicted in part (b) of Figure 1.2.

Through this example we can identify the following tasks of a compile-time garbage collection system:

- Clearly, such a system needs to determine at compile-time when particular data structures are accessed for the last time, after which they become dead and thus garbage. This detection can be done using the technique of program analysis, and more specifically, *abstract interpretation*, a technique that allows to derive run-time properties of programs without actually executing them. Here, the run-time property of interest is the knowledge about which data structures are dead, or the dual, which data structures remain *live*.

- Using the liveness information, one can thus detect when particular memory blocks become definitely garbage. It is then the task of the compile-time garbage collection system to determine what needs to be done with that garbage: either deallocate it or immediately reuse it?

- Once the compiler knows when data dies in a particular program, and how it can be reused, then the compiled code of that program should reflect these decisions and implement the actual memory reuse. In the case of our example, this means that for that particular call where the first argument is not used anymore further in the program, and where we want to reuse the

dead data locally, the compiler should generate code that performs a destructive update of the input argument.

- The compiler may be able to spot the possibility of a local reuse within the predicate of the example, yet whether or not the reuse is allowed depends on the caller of that predicate. Hence, the following questions must be answered:
  - Is it interesting to provide a variant of the original predicate that implements the detected memory reuse? Will there be any use for it?
  - When can calls to this predicate be replaced by calls to the optimised version, without compromising the safety of the program? How can we express the conditions with which we can guarantee this safety?
- Finding an answer to the previous questions becomes even more important in the presence of modules. Modules are often compiled separately, some modules may be even part of libraries, and thus pre-compiled w.r.t. the current project of the programmer. This means that during the compilation of one specific module, the compiler may not know how the entities defined in that module are used by the rest of the program. In our concrete example of the predicate updateSalary, this means that the compiler needs to detect that there is some potential of memory reuse of the first argument while being in some way *blind* about the way this predicate will be called by the remainder of the program. Moreover, even if the compiler does detect the memory reuse potential, how can that information be used?
- A CTGC system is clearly a complex system. Yet, to be usable in a real compiler, the slow-down of the overall compilation of a program must be in proportion to the obtained improved memory behaviour.

We give a brief overview of the work that already exists in some of these areas, then formulate the contribution of this work and finally, present the structure of this thesis.

1.3 Liveness Analysis

Mulkers (1991) defines live data structure analysis in the context of the logic programming language Prolog. It is based on the abstract interpretation framework of (Bruynooghe 1991) and essentially consists of two parts: a *structure sharing analysis* and the actual liveness analysis. The first part is a dedicated analysis that determines the possible memory sharing that may exist between the terms that are created during the execution of the program. This notion is not to be confused with *sharing analysis* (Bagnara, Zaffanella, and Hill 2005), an analysis whose aim

is to detect the sharing of the free variables within the constructed terms. Combining structure sharing information with the information about which variables are accessed during the forward execution of the program yields the description of all the memory blocks that this forward execution may possibly access. In this work backwards execution by means of backtracking is not specifically dealt with at analysis time. The author relies on the run-time system which must be enhanced such that memory reuse in the presence of backtracking remains safe. This means that the trailing mechanism must be adapted. Relying on the run-time system for this kind of safeness means an extra burden for the run-time system, reducing the overall win of using a compile-time garbage collector in the first place. In (Bruynooghe, Janssens, and Kågedal 1997) this problem is alleviated, and a formulation of the live structure analysis is presented that does explicitly take non-deterministic execution into account. The essential difference allowing this move is that in this work the underlying language is considered to be a logic programming language enriched with type, mode and determinism declarations. These declarations, absent in pure Prolog, are provided by the programmer as extra documentation of her/his code, yet at the same time, they allow the compiler to generate more efficient code. In this case, these declarations can be used as a form of added precision to the initial work of Mulkers (1991).

In the area of functional programming languages the terminology of *escape analysis* or *inheritance analysis* is often used. The aim of this analysis is the same: *obtain information on positions in a program where certain heap cells become obsolete during execution* (Mohnen 1995). Escape analysis allows for detecting which parts of the input to a function *escape* to the produced output. The parts that do not escape can safely be considered as garbage. The theory in (Mohnen 1997) is presented for a specific artificial and simplified functional language, and the escape information is formulated as a denotational characteristic of the language. An important aspect of escape information also consists of the sharing information between the manipulated data. Oddly enough, the author illustrates the use of the escape analysis for compile-time garbage collection, yet assumes that in such a setting, memory reuse is limited to structures which definitely will or can not share with other structures. Escape analysis has also been used in the context of object-oriented languages (Hill and Spoto 2002; Blanchet 1998; Blanchet 1999), yet there the prime intention of the analysis is to allocate objects of which the lifetime is known not to extend the lifetime of a given function on the stack instead of dynamically allocating them on the heap.

1.4 Reuse Decisions

The knowledge about terms becoming garbage at specific points in a program can be used in a number of different ways, regardless of the exact programming paradigm used:

- *Explicit Deallocation* (Hamilton 1995; Hamilton 1993). Here the data is immediately returned to the run-time system as soon as it has been detected as garbage. This approach is also used in (Hughes 1992), amongst others.
- *Destructive Allocation* (Hamilton 1995; Hamilton 1993). In this case, garbage is immediately reused for creating new structures. This approach is for example used in (Debray 1993; Gudjonsson and Winsborough 1993).

In (Gudjonsson and Winsborough 1993) the authors go even beyond simple reallocation of the dead data as their goal is to save every possible pointer or field update.

1.5 Modules

It is only recently that modules have been recognised as an element to be taken into account for the analysis of logic programs. Indeed, in the presence of modules, a program analysis system may either ignore the module structure of the program and analyse the source code stored in these modules as one single monolithic block, or conform to the usual compilation scheme of modules and thus analyse each module one at a time (Puebla and Hermenegildo 1999b). Obviously, the disadvantage of the first technique is that the analysis may have to be completely repeated whenever one of the modules changes. Moreover, the analysis of large programs may lead to resource problems on its own. These disadvantages are not present in the second compilation scheme. However, the analyses in such a setting need to be adapted such that they can still perform well in the absence of the complete source code of the program. Typically the complete lack of information of the other modules is alleviated by using dedicated *interface files* where intermediate analysis results of modules are stored for being used during the analysis of modules that depend on them. Depending on the complexity of the module structure of a program, more sophisticated analysis techniques may be needed (Bueno, García de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2001; Nethercote 2001).

1.6 Versions

Ideally, a predicate could be optimised for each specific call that can occur at runtime. If a predicate can be called in n different ways, then this means that potentially n optimised versions of the initial predicate code could be produced. While this ensures that every call to every predicate is indeed optimised, the effect of these optimisations may become negligible compared to the size of the obtained compiled code. This means that in practice a trade-off needs to be found between

the desired degree of program optimisation, and the code size that it may result in. A classic way of handling such situations is to generate a version for each particular call of a predicate, and to trim down this set of variants in a later step (Puebla and Hermenegildo 1999a; Vanhoof and Bruynooghe 1999; Leuschel, Martens, and De Schreye 1998). Another classic approach is to use some heuristic with which the analysis system can decide which versions are worthwhile, and which are not. This problem becomes even more important in the presence of modules, as in such a setting, the analysis system may not even know how some of the the entities defined in it are called. In such cases, a pure heuristics-based approach may be the only solution.

1.7 Mercury

Mercury is a modern logic programming language developed at the university of Melbourne, Australia (Somogyi, Henderson, and Conway 1996). This language was introduced with the intention to offer a pure declarative programming language — an ideal setting for automatic program transformations, optimisations, proof constructions, etc, while providing enough modern software engineering facilities to support the development of large *real-world* software applications. Moreover, an implementation of a Mercury compiler should guarantee the generation of reasonably fast and efficient programs. To meet these design objectives, Mercury uses a strong mode- and type-system, based on mode-, type- and determinism declarations that the programmer needs to provide for her/his code. These declarations are not only an essential form of documentation of the code, they also enable the compiler to perform a number of analyses that verify the correctness of these declarations, thus spotting a considerable number of bugs at compile-time. Moreover, these declarations enable the compiler to optimise and specialise the predicates for each of its declared uses, hence producing more efficient code. The support for programming in the large manifests itself by a modern module system using the notions of interfaces and implementations to distinguish public and private parts of a module.

The ongoing research around and within Mercury is still very active. This is shown by its involvement (Dowd, Henderson, and Ross 2001) with the .NET project (Microsoft ; Platt 2003). It also forms an interesting back-end for the constraint logic programming system HAL (Demoen, García de la Banda, Harvey, Marriott, and Stuckey 1999).

1.8 Goal

The goal of this thesis is to develop a complete compile-time garbage collection system based on the work of (Mulkers 1991), yet covering every aspect needed

for such a system as detailed above. The choice of Mercury as a target language is natural:

- Mercury relies on a good run-time garbage collector, yet given the purity of the language its memory demands remain high. The language provides the notion of *unique* objects (Henderson, Conway, Somogyi, and Jeffery 1996), but the use of these objects is limited due to a poor support for verifying their correctness.
- Given the extra declarations, the results of liveness analysis can be expected to be more precise than in its original setting, namely Prolog. This means that we can expect more opportunities for memory reuse in the setting of a compile-time garbage collection system implementing such a liveness analysis.
- The Mercury compiler available at this moment is mainly written in Mercury itself. This allows for a high-level declarative implementation of the CTGC system.

The choice of Mercury also brings its own challenges:

- Mercury is not Prolog. This means that the theoretical adaptation of the initial liveness analysis, and underlying structure sharing analysis, is not necessarily straightforward.
- Mercury programs are organised in modules, which is a challenge for program analysis in general, and for liveness analysis and the reuses one derives from it in particular.
- The CTGC system can be integrated into the Mercury compiler, yet should not slow down the compilation of the modules to a too great extent. Hence, the resulting CTGC system, to be usable in practice, must be fast and efficient, while remaining sufficiently precise to obtain the desired memory behaviour improvement.

1.9 Overview of the Thesis

In this thesis we define a number of semantics for the logic programming language Mercury, for which we use a denotational approach (Marriott, Søndergaard, and Jones 1994). We formally establish the equivalence between these semantics which enables us to correctly relate analysis results to concrete run-time properties that they are meant to approximate. We use these semantics to formalise each of the different analysis steps that are needed in a basic CTGC system. These analysis steps consist of a structure sharing analysis and liveness

analysis, inspired by (Mulkers 1991) and adapted to Mercury, followed by a so called reuse analysis, a new analysis designed to detect the actual memory reuses within a program.

Mercury programs are organised into modules, hence we extend this basic CTGC system such that it is able to correctly deal with a modular structure of the user program. This mainly poses a problem for the reuse analysis step as reuse can best be decided knowing the full calling context of the analysed predicates, which is not the case when analysing modules separately. Nevertheless, we successfully modularise the reuse analysis process.

At each stage of the formal development of the CTGC system we assess the feasibility of this system by implementing stand-alone prototypes. The positive results obtained with these systems motivate the development of a real CTGC system embedded into an existing compiler. We implement such a system in the Melbourne Mercury compiler. To the best of our knowledge, this is the first and only complete CTGC system that has ever been built for a programming language.

We now describe in some more detail the structure of the thesis.

After starting with some preliminary background in Chapter 2, we introduce Mercury in Chapter 3 and give it a formal syntax in Chapter 4. Given the differences between Prolog and Mercury, we define a new basic semantics of Mercury programs with which to express all the analyses required for our compile-time garbage collection system. Starting from a natural semantics, we derive a goal-independent based semantics using pre-annotations. The definition as well as the proofs of equivalence, are the subject of Chapter 5.

Using these semantics, we define structure sharing analysis for Mercury in Chapter 6. Chapter 7 describes the (near) syntactic properties of forward use and backward use. These notions are a key aspect in the definition of live structures. The reformulation of the liveness analysis is presented in Chapter 8. Chapter 9 introduces the notions of structure reuse, direct reuse, indirect reuse, notions which are indispensable in the characterisation of the memory reuse possibilities of a program. In that same chapter we also present some preliminary results of a prototype that estimates the potential of a CTGC system based on liveness analysis.

In Chapter 10 we introduce the notion of modules. While adapting the liveness analysis appears to be relatively easy, adapting the characterisations of the reuse possibilities is much less straightforward, ultimately leading to the notion of *reuse information* also called *reuse condition* which consists of a form of condition imposed on the calls of predicates to guarantee safe memory behaviour. We adapt the first prototype. The results are also presented in Chapter 10. Given the positive results obtained with this new prototype we move onwards, and implement a complete working CTGC system within the Melbourne Mercury compiler. The practical aspects leading to this implementation are detailed in Chapter 11. A number of benchmarks, ranging from small to medium-sized, are studied in

Chapter 12.

Finally we describe a first tentative approach to address the problem of characterising optimisations in such a way that more intelligent schemes can be found for deciding which versions of a specific predicate are worthwhile to generate, and which not. This is the subject of Chapter 13.

We conclude our work in Chapter 14, where we outline other yet not mentioned related research areas as well as interesting possibilities for future work in this interesting field of compile-time garbage collection for logic programming languages.

Chapter 2

Technical Background

In this chapter we give a brief overview of some technical background related to logic programming and its semantics.

2.1 Sets, Partially Ordered Sets, Complete Lattices

Let S be a set, then $\wp(S)$ is used in this thesis to denote the *powerset* of S. A *sequence* over S is an ordered list of elements of S. Sequences are written as: $a_1, \ldots, a_n$. We use $\overline{a}$ as a shorthand notation for a particular yet unspecified sequence of elements a. The set of finite sequences over S is written S^*.

We use $\mathbb{N}$ to denote the set of natural numbers.

A partial order defined for a set S is a binary relation, usually denoted using a comparison symbol such as $\leq$, $\subseteq$ or $\sqsubseteq$, that is reflexive, anti-symmetric and transitive. Thus, using $\leq$, $\forall x, y, z \in S$: $x \leq x$ (reflexive), $x \leq y \wedge y \leq x$ then $x = y$ (anti-symmetric) and $x \leq y \wedge y \leq z$ then $x \leq z$. A set S equipped with a partial order relation, say $\leq$, is called a *partially ordered set* (sometimes abbreviated to *poset*) and is usually denoted as $\langle S, \leq \rangle$. A poset $\langle S, \leq \rangle$ is called a *chain* if $\forall x, y \in S$, either $x \leq y$ or $y \leq x$. We say that $\langle S, \leq \rangle$ has a *bottom element* if there exists an element, usually denoted by $\bot$, such that $\bot \in S$, and $\forall x \in S : \bot \leq x$. Dually, S has a top elements, denoted by $\top$, if $\top \in S$ and $\forall x \in S : x \leq \top$. Note that finite chains always have bottom and top elements.

Two posets $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ may be related to each other by a function mapping elements in X to elements in Y. A map $\phi : X \to Y$ is said to be *monotonic* if $x_1 \leq_X x_2$ implies $\phi(x_1) \leq_Y \phi(x_2)$. If $\bot_X, \top_X$ and $\bot_Y, \top_Y$ are the bottom and top elements of X, resp. Y, then ϕ is called *strict* if $\phi(\bot_X) = \bot_Y$; it is called *co-strict* if $\phi(\top_X) = \top_Y$.

Introducing the *least upper bound*, also called the *join* operation, and *greatest*

lower bound, also called the *meet* operation, we obtain the notions of lattices and complete lattices. Using $\vee$ to denote the least upper bound operation in $\langle S, \leq \rangle$, then $x \vee y$, with $x, y \in S$, is defined as the least element of the set $\{s \mid x \leq s, y \leq s\}$. Dually, the meet of two elements in S, denoted as $x \wedge y$ is defined as the greatest element of the set $\{s \mid s \leq x, s \leq y\}$. Both notions are extended to sets of elements. Let S' be a subset of S, then $\bigvee S'$ denotes the least upper bound of the elements in S', and $\bigwedge S'$ denotes the greatest lower bound of these elements. A poset $\langle S, \leq \rangle$ is called a *lattice* if $\forall x, y \in S$ both $x \vee y$ and $x \wedge y$ exist. The poset is called a *complete lattice* if all subsets in S have a least upper bound as well as a greatest lower bound. Complete lattices based on a poset $\langle S, \leq \rangle$ with least upper bound $\vee$, greatest lower bound $\wedge$, bottom element $\bot$ and top element $\top$ are usually denoted using the tuple $\langle S, \leq, \vee, \wedge, \bot, \top \rangle$. When $\subseteq$ is the underlying ordering, we usually write $\langle S, \subseteq, \cup, \cap, \bot, \top \rangle$, and with $\sqsubseteq$ we use $\langle S, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$.

Let ϕ be a map from elements in a complete lattice $\langle X, \subseteq, \cup, \cap, \bot, \top \rangle$ to elements in a complete lattice $\langle Y, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$, then ϕ is said to be *continuous* if for all subsets D in X, we have: $\phi(\bigcup D) = \bigcup\{\phi(x) \mid x \in D\}$. Usually, the latter expression is abbreviated to $\bigcap \phi(D)$. Complete lattices in which every ascending chain is finite are called *Noetherian*. Such domains will be used in the definition of our formal semantics of the Mercury language.

These notions will be essential when proving the well-definedness of the semantics given for the Mercury language.

For more details we refer the reader to (Davey and Priestley 2002; Nielson, Nielson, and Hankin 1999).

2.2 Logic Programming

We give a brief overview of the basic elements of logic programming. For more details we refer the reader to (Lloyd 1987).

The basic alphabet of a logic program consists of a finite set of variables $\mathcal{V}$, a finite set of function symbols Σ and a finite set of predicate symbols Π. Function and predicate symbols are associated with an *arity*, a natural number identifying the number of *arguments* the function or predicate symbol has. Function symbols with arity zero are called *constants*. We use the following notation:

- uppercase letters, e.g. X or Y, or capitalised words, e.g. *First* or *Tail*, are used for elements from $\mathcal{V}$,*i.e.*, variables;
- f, g, h denote function symbols;
- p, q usually denote predicate symbols.

If the arity of the function or predicate symbol is of importance, we explicitly write f/n or p/m to denote the function symbol f or predicate symbol p with arities $n, m \in \mathbb{N}$.

A *term* is a variable or a compound term $f(\tau_1, \ldots, \tau_n)$ where $f/n \in \Sigma$ and each argument τ_i is a term. The set of terms is denoted by $\mathcal{T}(\mathcal{V}, \Sigma)$. A term is usually denoted using the character τ, possibly sub- or superscripted. An *atom* is a predicate symbol $p/n \in \Pi$ applied to a sequence of n terms. The predicate symbol $=/2$ (usually written in infix notation) is treated in a special way and is called an *explicit unification*. We use the notion of *expression* to refer to either a term, an atom, or any composition of these elements.

A *ground term* is a term that does not contain any variables. A *ground atom* is an atom not containing any variables. Let a be any syntactic object of the language, then we use $Vars(a)$ to denote the set of variables occurring in that object. If a is known to contain at most one variable, then $Var(a)$ is used to denote that variable.

2.3 Variable Substitutions

A *substitution* θ is a finite mapping from distinct variables to terms: $\mathcal{V} \to \mathcal{T}(\mathcal{V}, \Sigma)$. As usual (Lloyd 1987), we represent substitutions as:

$$\theta = \{X_1/\tau_1, \ldots, X_n/\tau_n\}$$

where each $X_i \neq \tau_i$. Each element X_i/τ_i is called a *binding*. θ is called a *ground* substitution if each of the terms $\tau_i, 1 \leq i \leq n$ is ground.

If E is an expression and θ a substitution, then $E\theta$ is the expression obtained from E by simultaneously replacing each occurrence of a variable X_i in E by the term τ_i, where $X_i/\tau_i \in \theta$. $E\theta$ is called an *instance* of E. We say that θ is *applied* to the expression E. If $E\theta$ is ground, then it is called a *ground* instance of E. We then say that θ grounds the expression E. These notions are generalised to sets of expressions.

If θ is the empty set, then it is called the *identity substitution*.

If E and F are expressions, then E and F are called *variants* if there exist substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. A *renaming substitution* for an expression E is a variable pure substitution $\{X_1/Y_1, \ldots, X_n/Y_n\}$ such that $(\{Y_1, \ldots, Y_n\} \setminus \{X_1, \ldots, X_n\}) \cap Vars(E) = \{\}$. Two expressions E and F are *equivalent up to renaming*, written $E \sim F$, if $E\rho = F$ for some renaming ρ. The equivalence class of E under $\sim$ is denoted by $[E]_\sim$. All the elements in $[E]_\sim$ are variants of each other.

2.4 Existentially Quantified Term Equations

Usually, substitutions are used to represent the computed answers of a logic program. In this thesis we choose the domain of constraints to express this informa-

tion for the same reasons as Marriott, Søndergaard, and Jones (1994), namely for the simplicity of expressing and ordering constraints.

Here, variable bindings generated by a logic program are seen as *constraints* or so called *existentially quantified term equations* (Jaffar and Maher 1994; Marriott, Søndergaard, and Jones 1994; García de la Banda, Marriott, Stuckey, and Søndergaard 1998).

Definition 2.1 (Ex-equation) *(Marriott, Søndergaard, and Jones 1994) An* ex-equation *is a possibly existentially quantified conjunction of basic equations $T_1 = T_2$, where $T_1, T_2 \in \mathcal{T}(\mathcal{V}, \Sigma)$. The conjunction may be empty, in which case we denote it by* true*. The set of ex-equations is called Eqn.*

Definition 2.2 (Satisfiable, solvable ex-equation) *An ex-equation is said to be* satisfiable *or* solvable *in the algebraic sense of the word,* i.e.*, if each variable appearing in the equation can be given a value such that every constraint of the ex-equation is satisfied.*

Definition 2.3 (Solved form) *An ex-equation e is in* solved form *if it is satisfiable and if each of the basic equations is of the form $X = t$ where $X \in \mathcal{V}$ and $t \in \mathcal{T}(\mathcal{V}, \Sigma)$.*

How to obtain the solved form of an ex-equation is irrelevant for this thesis. We refer the reader to (Martelli and Montanari 1982) amongst others.

Let $Eqn^+ \subseteq Eqn$ be the set of all satisfiable ex-equations in Eqn.

Example 2.1 *The following are elements in Eqn:*

$X = Y$	$X = f(Y) \wedge Z = g(T)$
$\exists Y \,.\, X = f(Y)$	true
$X = 1 \wedge X = 2$	false

Only the first two rows contain satisfiable ex-equations. The last two constraints are not in Eqn^+.

Definition 2.4 (Ordering in Eqn, Eqn^+) *The elements of Eqn are ordered by the logical consequence operator $\models$.*

Two ex-equations $e_1, e_2 \in Eqn$ (or similarly in Eqn^+) are called *equivalent*, which is written as $e_1 \equiv e_2$, iff $e_1 \models e_2$ and $e_2 \models e_1$. The equivalence class of an ex-equation e under $\equiv$ is denoted by $[e]_\equiv$. Partitioning Eqn under this equivalence definition, we obtain a complete lattice with $[\mathsf{true}]_\equiv$ as the greatest element and $[\mathsf{false}]_\equiv$, the equivalence-class of unsatisfiable elements, as the smallest. Partitioning Eqn^+ in the same way, we obtain that Eqn^+ is a partially ordered set with only a greatest element, namely $[\mathsf{true}]_\equiv$, and no bottom element.

In this text we mainly deal with the equivalence classes of ex-equations, and therefore we abbreviate our notation, and simply assume that when handling an

ex-equation we are in fact handling the equivalence class that the ex-equation is an element from. Hence, true will be used instead of $[\mathsf{true}]_{\equiv}$, and in general e is used instead of $[e]_{\equiv}$, $\forall e \in Eqn$ or $\forall e \in Eqn^{+}$.

Example 2.2 *Let* $e_1 = (X = Y)$ *and* $e_2 = (X = f(Z) \wedge Y = f(Z))$ *then* $e_2 \models e_1$. *But also,* $\mathsf{false} \models e \models \mathsf{true}$, $\forall e \in Eqn$.

The relation between variable substitutions and ex-equations is given by the notion of *unifier*:

Definition 2.5 (Unifier) *A* unifier *of a constraint* $e \in Eqn$ *is a substitution* θ *such that* $\mathsf{true} \models e\theta$. *The set of unifiers of a constraint* e *is denoted by* $\mathsf{unif}(e)$.

Note that the identity substitution can also be a unifier for an ex-equation.

Example 2.3 *Let* $e = (A = B)$, *and*

$$\begin{array}{lcl} \theta_1 & = & \{A/X, B/X\} \\ \theta_2 & = & \{A/f(T), B/f(T)\} \\ \theta_3 & = & \{A/3, B/3\} \\ \theta_4 & = & \{A/f(1), B/f(Y)\} \end{array}$$

Here θ_1, θ_2 *and* θ_3 *are unifiers for* e, *while* θ_4 *is not. Indeed* $e\theta_4 = (f(1) = f(Y))$ *is not true in all models of* true.

Definition 2.6 *An ex-equation* e *is said to* ground *a variable* X *iff all unifiers of* e *are ground substitutions for* X.

Example 2.4 *Consider the constraint* $e = (X = f(Y) \wedge Y = 3)$, *then* $\mathsf{unif}(e) = \{\{Y/3, X/f(3)\}\}$ *which means that* e *grounds both* X *and* Y.

Definition 2.7 (Ordering in $\wp(Eqn)$ and $\wp(Eqn^{+})$) *The domain of sets of (equivalence classes of) equations,* $\wp(Eqn)$ *resp.* $\wp(Eqn^{+})$, *is ordered by the usual set ordering* $\subseteq$. *It has a least upper bound* $\cup$, *greatest lower bound* $\cap$, *least element* $\{\,\}$ *and greatest element* Eqn, *resp.* Eqn^{+}.

In the remainder of this thesis we shall only use $\wp(Eqn^{+})$ as our domain of expressing variable bindings in the user program. This is not a restriction. Indeed, while all unsatisfiable constraints obtained in $\wp(Eqn)$ are kept in the equivalence class represented by false, in $\wp(Eqn^{+})$ such elements are simply discarded. This simplifies most of the operations that we will define on $\wp(Eqn^{+})$ and also removes the possibly confusing equivalence of $\{\,\}$ and $\{\mathsf{false}\}$ in $\wp(Eqn)$.

Note that both $\wp(Eqn)$ and $\wp(Eqn^{+})$ are complete lattices. Their ordering is only related to the ordering in Eqn, resp. Eqn^{+} by the equivalence relation that the latter ordering implies.

We introduce two operations: projection and renaming. The purpose of projecting an equation e onto a set of variables V is to obtain a new equation e' such that $Vars(e') \subseteq V$ and $e \models e'$. Throughout this thesis we use the notation $(a)|_V$ to project the information represented by the object a onto the set of variables V. Of course, the exact definition will be different for each type of object a. The goal of renaming an equation e is to replace each occurrence of a variable in that equation by another variable. Renamings are usually denoted by $\rho_{s_1 \to s_2}(a)$ where a is any object of interest, and $s_1, s_2 \in \mathcal{V}^*$.

The specific definitions of projection and renaming for ex-equations and sets of ex-equations are given below.

Definition 2.8 (Projection) *Let e be an ex-equation in general (thus in Eqn) then projecting e on a set of variables V is defined as:*

$$(e)|_V = \overline{\exists}_V.e$$

where $\overline{\exists}$ is the existential quantification w.r.t. the complement of the variables V, i.e., *$\mathcal{V} \setminus V$. For $E \in \wp(Eqn)$, the projection is defined as:*

$$(E)|_V = \{\overline{\exists}_V.e \mid e \in E\}$$

Let S be some syntactic construct within our language, e.g. an atom or term, then $(E)|_S$ is used as a shorthand notation for $(E)|_{Vars(S)}$.

Example 2.5 *Let $E = \{X = f(Y) \wedge Y = Z\}$. The projection onto the variables $\{X, Z\}$ is $(E)|_{\{X,Z\}} = \{\exists Y.X = f(Y) \wedge Y = Z\}$ which is equivalent to $\{X = f(Z)\}$.*

Definition 2.9 (Ex-equation Renaming) *Let $e = (T_1 = T_1' \wedge \ldots \wedge T_n = T_n') \in Eqn$, then renaming e with respect to a mapping between a sequence of variables $X_1, \ldots, X_n$ and a sequence of variables $Y_1, \ldots, Y_n$, with $(\{Y_1, \ldots, Y_n\} \setminus \{X_1, \ldots, X_n\}) \cap Vars(E) = \{\}$, is defined as:*

$$\rho_{\overline{X} \to \overline{Y}}(e) = (T_1\theta = T_1'\theta \wedge \ldots \wedge T_n\theta = T_n'\theta)$$

where θ is the renaming substitution $\{X_1/Y_1, X_2/Y_2, \ldots, X_n/Y_n\}, 1 \leq i \leq n$.

The definition is extended to elements of $\wp(Eqn)$ in a natural way. Let $ES \in \wp(Eqn^+)$:

$$\rho_{\overline{V_1} \to \overline{V_2}}(E) = \{\rho_{\overline{V_1} \to \overline{V_2}}(e) \mid e \in ES\}$$

By abuse of notation, we sometimes use renamings in the context of sets of variables instead of explicit sequences. In that case we assume that the ordering of the elements of these sets and the mapping is naturally implied by the context. For example, if τ_1 and τ_2 represent two terms with variable arguments, then the variables of the terms in the renaming $\rho_{Vars(\tau_1) \to Vars(\tau_2)}(E)$ are simply ordered by their position in the terms. We even abbreviate such a renaming to $\rho_{\tau_1 \to \tau_2}(E)$.

Example 2.6 *Let* $E = \{X = f(Y) \wedge Y = Z, X = g(3) \wedge Y = Z\}$, *let* S_1 *be the sequence* X, Y, Z *and* S_2 *the sequence:* U, V, W, *then* $\rho_{S_1 \to S_2}(E) = \{U = f(V) \wedge V = W, U = g(3) \wedge V = W\}$. *Let* $\tau_1 = h(X, Y, Z)$, *and* $\tau_2 = h(U, V, W)$, *then* $\rho_{\tau_1 \to \tau_2}(E) = \rho_{S_1 \to S_2}(E)$.

Chapter 3

Mercury

Mercury is a pure functional logic programming language conceived and developed at the University of Melbourne, Australia. It was introduced in 1993 with the intention to offer a *pure* logic (functional) programming language, carefully avoiding the typical pitfalls of the common logic languages (e.g. Prolog (Sterling and Shapiro 1986)) which are their bad run-time performance, and poor support for *programming in the large*. Mercury was therefore developed with the following clear characteristics in mind:

- Mercury should be a *pure declarative language*. Typically Prolog uses non-pure language constructs such as *assert*, *retract*, or others. These constructs allow a local performance gain, but inflict an extra burden on the programmer as she/he is now expected to take into account low-level performance criteria instead of purely focusing on the high-level program design aspects of his project.

- Mercury should provide explicit support for developing programs in the context of a *team of programmers*. The now widely accepted technique is the use of *modules*. These modules provide clear interfaces and hide all implementation details. Modules are usually compiled separately.

- Unlike most Prolog systems which merely check the syntax of the programs and which only give sparse and obscure error messages, a Mercury compiler should (try to) deliver *strong error messages*, hence the programs should contain sufficient redundant information that can be verified and checked by a compiler.

- And finally, Mercury programs should be compiled to *fast and efficient programs*. Their performance should at least be as efficient as comparable programs in comparable languages, but should in preference be better.

The result is a highly competitive pure functional logic programming language with support for a number of modern software engineering concepts such as modules, type classes, higher order logic, and which, thanks to program declarations, enables remarkable performance. We give a brief summary of the main elements of the Mercury programming language. We refer the reader to (Henderson, Conway, Somogyi, and Jeffery 1996) for a full presentation of the language.

The formal notations introduced in this section are inspired by the presentation of Mercury given in (Vanhoof 2001).

3.1 Predicate Clauses

Predicate clauses in Mercury are similar to Prolog clauses. Each clause consists of a head atom and a body. A body is simply a goal. A goal can either be a conjunction of goals, a disjunction of goals, an if-then-else construct, a negated goal, or simply a literal. A literal is either an explicit unification or an atom. If the goal is an empty conjunction (which is a goal that always succeeds), then the clause is called a *fact*.

Mercury has a functional flavour in the sense that programmers are allowed to declare and use functions instead of predicates.

Example 3.1 *The clauses for a predicate defining the concatenation of two lists are:*

```
append([ ], Y, Y).
append([Xe|Xs], Y, [Xe|Zs]):- append(Xs, Y, Zs).
```

In Prolog, the clauses would be the same.

Example 3.2 *Adding two boolean values can be defined by a function* add/2 *described by the following two clauses:*

```
add(true, true) = true.
add(false, _) = false.
add(_, false) = false.
```

In the remainder of this thesis we do not distinguish functions from predicates as the former can always be transformed into the latter by augmenting the arity of the function by one, and using the new argument to represent the result of the original function.

Example 3.3 *The* add/2 *function defined in the previous example can be rewritten as a predicate defined by the following facts:*

```
add(true, true, true).
add(false, _, false).
add(_, false, false).
```

A full Mercury program is required to declare and implement a predicate main/2. This is the *entry point* to the program. After compilation, the execution of the program always starts at this entry point. This is similar to the main-function that needs to be implemented in C-programs (Kernighan and Ritchie 1978).

3.2 Type Declarations

Mercury is a strongly typed language. Its type system is based on a polymorphic many-sorted logic, inspired by the Hindley-Milner (Hindley 1969; Milner 1978) type system of ML (Milner, Tofte, and Macqueen 1997).

The type system consists of type declarations defining the types introduced by the programmer, and type declarations of the predicates and functions defining the types of the arguments used in the predicates and functions.

Basically each type is declared as a discriminated union, *i.e.*, a set of function symbols and the types of their arguments. A type can also be declared as equivalent to another type, yet we see this as syntactic sugar that allows the programmer not to repeat a type definition.

Example 3.4 *The following declarations are valid type declarations in Mercury:*

```
:- type boolean ---> true; false.
:- type list(T) ---> []; [T|list(T)].
:- type intlist == list(int).
```

The types boolean *and* list(T) *are defined as discriminated union types. Type* intlist *represents lists of integers, and is defined as being equivalent to the* list*-type, where the type-variable is initialised with the (built-in) type* int *(representing integers).*

Formally, let $\Sigma_{\mathcal{T}}$ denote the set of type constructors. Each type constructor is associated with a natural number called the *arity* of the type constructor. Let $\mathcal{V}_{\mathcal{T}}$ denote the set of type variables. The set of types is then represented by $\mathcal{T}(\Sigma_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}})$. In a formal setting, type constructors are written in a sans serif font, e.g., t, and with its explicit arity t/n. In examples of real source code, no special font is used. A type containing variables is said to be *polymorphic*, otherwise it is called *monomorphic*. A *type substitution* is a substitution mapping type variables to types. Applying a type substitution to a polymorphic type results in a new type, called an *instance* of the original type. We say that a type $\mathsf{t}_1/n \in \Sigma_{\mathcal{T}}$ *matches* with a type $\mathsf{t}_2/n \in \Sigma_{\mathcal{T}}$ if t_1/n is an instance of t_2/n. Overloading the substitution notation we may write this as: $\mathsf{t}_1 = \mathsf{t}_2\theta_{\mathcal{T}}$, where $\theta_{\mathcal{T}}$ is a type substitution.

Example 3.5 *In the type declarations in Example 3.4, we have*

$$\{\mathsf{boolean}/0, \mathsf{list}/1, \mathsf{intlist}/0, \mathsf{int}/0\} \subset \Sigma_{\mathcal{T}}$$

and $\mathsf{T} \in \mathcal{V}_{\mathcal{T}}$*. All the types are monomorphic, except* list(T)*. The type* list(int) *is an instance of* list(T)*, with type substitution* $\{\mathsf{T}/\mathsf{int}\}$*.*

Definition 3.1 (Type Declaration) *A* type declaration *associated with a type constructor $h/n \in \Sigma_{\mathcal{T}}$ is a definition of the form*

$$\mathsf{h}(\mathsf{T}_1, \ldots, \mathsf{T}_n) \rightarrow c_1(\overline{\mathsf{t}_1}); \ldots; c_m(\overline{\mathsf{t}_m}).$$

where $\{\mathsf{T}_1, \ldots, \mathsf{T}_n\} \subseteq \mathcal{V}_{\mathcal{T}}$, $c_i/k \in \Sigma$ for $1 \leq i \leq m$, with $\overline{\mathsf{t}_i}$ a sequence of types from $\mathcal{T}(\Sigma_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}})$. Also $\bigcup\{Vars(c_i(\overline{\mathsf{t}_i})) \mid 1 \leq i \leq m\} \subseteq \{\mathsf{T}_1, \ldots, \mathsf{T}_n\}$. The function symbols $c_1, \ldots, c_m$ are said to be associated *with the type constructor h/n.*

In actual programs, we use the symbol `--->` instead of $\rightarrow$ in type declarations. The latter is only used in formal settings.

In theory, every type can be defined by a type declaration. In practice, Mercury provides a number of *built-in* types. These types are `int`, `float`, `char` and `string`, representing resp. the set of integers, floating point numbers, characters and strings. Given the importance of the memory-representation of terms in this thesis, but also given the fact that this representation is strongly implementation dependent we present these low-level issues in a separate section (Section 3.8).

Beside the type declaration defining individual types, Mercury requires each exported[1] predicate or function to be accompanied with a definition of the types of the arguments it uses. The compiler should be able to some extent to infer this information automatically for the other predicates. Yet, the programmer is strongly encouraged to provide all the extra redundant information. This enables a compiler to verify this extra information and compare it with the inferred information (hence verifying the intended correctness). Moreover, from a software engineering point of view this consists of valuable documentation.

Example 3.6 *The following are valid predicate/function type declarations in Mercury:*

```
:- pred append(list(T), list(T), list(T)).
:- func and(boolean, boolean) = boolean.
```

The first line declares that the program provides a predicate named `append` *with arity 3, and of which all arguments are of the polymorphic type* `list(T)`*. The second line declares the types of the function* `and/2`*: it takes two* `boolean` *arguments, and yields a* `boolean` *result.*

Using the type declarations, and predicate/function type declarations, the Mercury compiler is capable of inferring the exact type for each variable occurring in the program. In the same time Mercury compilers are supposed to verify whether the program is *well typed or Hindley/Milner type correct* (Mycroft and O'Keefe 1984; Pfenning 1992). Programs that are not type correct must be rejected.

[1] exported w.r.t. a module, see Section 3.5.

Example 3.7 *Type correct programs that contain calls to* append *as declared in Example 3.6 guarantee that each of these calls is done with types matching the declared type. Therefore calling* append *with all three arguments being* list(int) *is perfectly legal. Note that calling* append *with mixed arguments, say* list(int) *and* list(boolean) *is not legal.*

Mercury also supports type classes based on the type classes used in the lazy functional programming language Haskell (Hudak et al. 1992). A type class provides a set of names and signatures of class operations. A type can be made an instance of a type class by explicitly instantiating each of the declared operations of that type class with a specific implementation that can be used for that type. The use of type classes in a program enables the programmer to define predicates that are not only parametrised w.r.t. the types of the terms that they act on (which can be achieved through the usual polymorphism of the language) but also w.r.t. operations defined on these types. As such, type classes introduce a form of higher-order programming. Given the fact that our analyses are mainly oriented towards first-order logic, we do not sketch any further details about type classes and their use in Mercury.

3.3 Mode Declarations

The mode information of a predicate (or function) describes the mapping from the initial instantiation of the arguments of the predicate (or function) to their final instantiation. Instantiation states are described using type information. Viewing each type as a regular tree with or-nodes representing types and and-nodes representing type constructors, an instantiation state of a procedure argument describes the instantiation of each of the or-nodes of its type. Mercury uses two base cases of instantiation states, namely `free` — the argument is a free variable, and `bound` to a specific constructor — the argument is `bound` to a term with that specific constructor. These two cases can be combined to express more refined instantiation states, like for example:

```
:- inst listskel == bound( []; [free | listskel] ).
```

An argument with this instantiation is either bound to the constant `[]`, or to a term with outermost functor `[|]`, its first argument free, and its second argument also corresponding to the listskel instantiation state. As a shorthand notation Mercury provides the instantiation state ground which describes argument being bound to fully ground terms. Of course, the exact instantiation state depends on the type of the argument to which it applies.

Instantiation states are used to describe modes. A mode is a mapping from an initial instantiation state to a final instantiation state. Mercury provides two standard modes, `in` and `out`, which are defined by the rules:

```
:- mode in  == ground >> ground.
:- mode out == free >> ground.
```

If a procedure argument has mode `in`, then this means that when the procedure is called, that particular argument will be bound to a ground term and remains bound to a ground term upon exit from the procedure call. If an argument has mode `out` then the argument will be a free variable when entering the procedure, yet it will become fully instantiated when leaving the procedure call. Of course, the user is free to define more complicated modes.

Using these modes, the programmer can document her/his predicates and functions with *mode declarations*. These declarations define a valid combination of modes for the arguments of the predicate. In logic programs it is common to use the same predicate in different ways, which in Mercury becomes explicit through the definition of multiple mode declarations for that same predicate. Note that like for predicate declarations, mode declarations are mandatory for exported predicates and functions, and only recommended otherwise.

Mercury is a strictly moded language. This means that it does not allow the use of *partially instantiated structures* unless of course it corresponds to the instantiation state with which it was declared.

Example 3.8 illustrates the syntax of these mode declarations.

Example 3.8 *As a complement to the predicate and function type declarations of Example 3.6 the programmer may provide the following mode declarations:*

```
:- mode append(in,in,out).
:- mode append(out,out,in).
:- mode add(in,in) = out.
```

This declares two modes for append/3*. The first mode states that* append *changes the instantiation state of the third argument to ground (mode* `out`*) if it is called with its two first arguments ground (mode* `in`*). The second mode states that* append *can also be used if only the third argument is ground at call time. Upon success, the two first arguments will be bound to a ground term. The mode declaration for* add/2 *can be interpreted similarly.*

In the previous example, two mode declarations were given for the same predicate. Although it defines two uses of the append predicate, the programmer need only write the implementation of append once. It is up to the compiler to generate the two versions for the two uses of this predicate (if of course these versions differ).

In general, each mode of a predicate may require a different compiled version of that predicate. For this purpose, the notion of a *procedure* is used.

Definition 3.2 (Procedure) *The set of clauses defining a predicate and a specific mode declaration for that predicate is called a* procedure.

As argued above and as will be illustrated further on, each procedure may be compiled to a separate version of the initial predicate definition it stems from. The specific version implementing the actual procedure is called the *procedure definition*.

In the further text we implicitly assume that every procedure has its own definition, and therefore we use the terms *procedure* and *procedure definition* interchangeably. Literals within a procedure definition that are not calls to built-in operations are called *procedure calls*.

3.3.1 Specialised Unifications

As the instantiation state of a variable in a procedure can perfectly be determined using the present mode information, each unification can be specialised in either a *construction* — `X <= f(Y1,...,Yn)`, a term `f(Y1,...,Yn)` is built and assigned to the variable `X`, a *deconstruction* — `X => f(Y1,...,Yn)`, the term to which variable `X` is bound is decomposed, a *test* — `X == Y`, the terms pointed at by the variables `X` and `Y` are compared w.r.t. syntactic equality, and finally an *assignment* — `X := Y`, variable `X` is bound to the term pointed at by variable `Y`.

The details of these four different unifications are given in Section 4.2.3.

3.3.2 Selection Strategy

Mercury is a strictly moded language where each literal may only be called with arguments having the instantiation states as given by the mode declarations. This automatically imposes a selection strategy for the language as a literal can only be called if its arguments are sufficiently instantiated. If the run-time system has the choice of evaluating one literal or another, then it has to select the literal that appears first in the sequence of literals defining that procedure. This strategy is usually called the *left-to-right selection strategy* and is one of the most common strategies used in logic programming in general.

Given the fact that mode information is available during the compilation of a procedure (either because it was provided by the user, or inferred by a separate mode inference engine), the compiler can perfectly determine the order of execution of the literals in each of the procedures of the user program and can therefore rearrange the literals of a procedure definition accordingly.

Example 3.9 shows how the general definition of `append/3` can be specialised for the two modes given in Example 3.8.

Example 3.9 *Consider the predicate and mode declaration for* `append`*, as well as its general definition:*

```
:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out).
```

```
:- mode append(out,out,in).
append([],Y,Y).
append([Xe|Xs],Y,[Xe|Zs]):- append(Xs,Y,Zs).
```

For each mode declaration, a Mercury compiler will produce a separate procedure with its own definition. Assuming the left-to-right selection strategy, theses procedure definitions may look as follows:

- *Version for the mode* `(in, in, out)`*:*

```
append(X,Y,Z):-
        (
          X => [ ], Z := Y
        ;
          X => [Xe|Xs], append(Xs,Y,Zs), Z <= [Xe|Zs]
        ).
```

- *Version for the mode* `(out, out, in)`*:*

```
append(X,Y,Z):-
        (
          X <= [ ], Y := Z,
        ;
          Z => [Xe|Zs], append(Xs,Y,Zs), X <= [Xe|Xs]
        ).
```

The first version differs from the second version mainly by the ordering of the literals appearing in the second branch of the disjunction.

Note that here we have introduced explicit disjunctions instead of producing two separate clauses per predicate and that all general unifications are replaced by their specialised forms.

In the remainder of the thesis we assume that each of the procedures is *well moded* w.r.t. the left-to-right selection strategy. Well modedness is defined as follows:

Definition 3.3 (Well modedness) *A procedure call is said to be* well moded *if, at the time the procedure is called, the arguments have the correct instantiation w.r.t. the modes declared or inferred for that procedure.*

A compiler may thus have to rearrange the individual literals within a procedure definition in order to achieve well modedness. If the compiler can not find an adequate rearrangement that does not violate the modes of the called procedures, then the compiler must reject that procedure, hence also the program. A program is considered *well moded* if all the procedure calls within it are well moded.

3.3.3 Unique modes.

In Mercury it is also possible to add *uniqueness* information to the usual changes of instantiation captured by the basic `in` and `out` modes. It provides the additional built-in instantiation states `unique` and `dead`. An argument of a literal is said to be `unique` if the term to which it is bound is ground and if that argument is the *only* reference to that specific term. The instantiation `dead` means that there are no references to the value an argument pointed to in which case the compiler is free to generate code reusing that value. The main modes defined on these uniqueness instantiation states are `di` — *destructive input* — and `uo` — *unique output*:

```
:- mode di == unique >> dead.
:- mode uo == free >> unique.
```

The arguments declared as `di` of a procedure call must guarantee that they have a unique reference to the term they are bound to and that these arguments will not be used after that procedure call, hence, that unique reference becomes a dead pointer. Arguments declared as `uo` are free arguments when entering a procedure call, and are guaranteed to be bound to a ground term to which they have a unique reference.

The use of these modes is not popular given the fact that it adds a clear low-level aspect to the programming task whereas the programmer is encouraged to mainly focus on high-level abstractions made possible within the declarative programming paradigm. Also, the task of verifying the correctness of these modes is cumbersome, which means that, at the current state of the compiler, the analysis for verifying the use and propagation of unique modes remains rather conservative: it can only prove correctness of the use of these modes in the simplest cases.

Given the mentioned difficulties, unique modes are mainly used for giving a declarative meaning to input/output operations (which operate on a unique state of the world) or for performance critical data structures such as arrays. Such operations and data structures are mainly defined in the standard library of the language and are usually implemented in a foreign language such as C. In such cases, these unique mode declarations become valuable information for the compile-time garbage collection system developed in this thesis as this system is unable to verify code written in a foreign language. Note that even the unique modes analyser within the compiler itself does not analyse foreign code and therefore assumes that the annotations are correct. See also Section 11.2. Moreover, the direct use of these modes in user-programs (hence direct memory management by the programmer) becomes to some extent obsolete in the presence of the compile-time garbage collection system that is developed in this thesis.

The definition of the Mercury language allows the possibility of defining more elaborate uniqueness states and modes as given by this overview such as poly-

Can fail	Number of solutions	Determinism
yes	0	`failure`
no	0	`erroneous`
yes	1	`semidet`
no	1	`det`
yes	≥ 1	`nondet`
no	≥ 1	`multi`

Table 3.1: Mercury determinism information. The first column describes whether it can fail or not. The second column determines the number of solutions the procedure may have.

morphic modes or mostly unique modes, yet these modes are not of great importance in this thesis.

3.4 Determinism Declarations

Besides type and mode information, each procedure can also be given determinism information. This information describes whether the procedure can fail or succeed. If it can succeed, then it also describes the number of solutions that can be found. This information can be provided by the programmer (mandatory for exported predicates) or inferred by the compiler. The determinacy of a procedure is declared together with its mode information.

Table 3.1 lists the determinacy options provided in Mercury. The most important determinism characterisations are `semidet`, `det`, `nondet` and `multi`. The descriptions `failure` and `erroneous` are used rarely and should be seen as exceptional.

The determinacy of a procedure is declared together with its mode information as illustrated in Example 3.10.

Example 3.10 *Completing the mode declarations from Example 3.8 with determinism information, we have:*

```
:- mode append(in,in,out) is det.
:- mode append(out,out,in) is multi.
:- mode add(in,in) = out is det.
```

The first line not only describes a mode for append*, but also declares that when* append *is called with its two first arguments ground lists, and last argument a free variable, then the procedure is deterministic: exactly one solution is produced, it can not fail. The second mode of* append *is declared as* `multi`*: it can produce multiple solutions, yet it can not*

fail either. Determinism information can also be given for functions, although, if omitted, the Mercury language assumes them to be deterministic.

Mercury also allows *committed choice* non-determinism where the compiler is informed about the fact that a procedure may in general have multiple solutions, yet where the caller of that procedure may be interested in exactly one of its solutions without minding which one exactly. These forms of determinism are variations on the usual ones, hence, without loss in generality, we do not consider these annotations in this thesis.

3.5 Modules

Mercury programs are divided into *modules*. Each module consists of an *interface* section and an *implementation* section. Types, modes, predicates, functions that are declared in the interface of a module are made visible to those modules that import this module. The entities declared in the interface section of a module are said to be *exported*: *exported predicates*, *exported types*, etc. Everything that is declared in the implementation section of a module is visible only within that module. These entities are *hidden*. Predicate and function clauses are only allowed in the implementation section.

It is possible to declare the name of a type in the interface section while the full declaration is only given in the implementation. Such types are called *abstract types*.

When a module wants to use the exported entities of another module, the former needs to *import* the latter. This is done by the `import_module` directive.

Example 3.11 *The use of modules is illustrated by the program code shown in Figure 3.1 defining the module* `app` *that imports the (built-in) module* `list`*—a module that defines all list-related types and predicates.*

3.6 Higher-Order Language Features

Mercury supports higher order programming. A higher order term can for example be constructed as follows:

```
Prepend = pred(I::in, O::out) is det :- append(LL,I,O)
```

This unification constructs a deterministic closure with arguments `I` and `O` having mode `in` resp. `out`, and binds it to the variable `Prepend`. The variable `LL` is a variable that needs to appear in the same scope as the higher order term.

Closures are invoked with the `call/N` predicate where `N` is a variable number of arguments corresponding to the arity of the closure. Using the above example

```
:- module app.

% Interface-section.
:- interface.

% Imported modules.
%   Module "list" defines the
%   type "list(T)" and the
%   predicate "append/3".
:- import_module list.

% Declaration of the abstract
% type 'thing'.
:- type thing.

% Procedure declarations.
:- pred thingy(int, thing).
:- mode thingy(in, out) is det.
:- mode thingy(out, in) is det.
:- pred concat(list(thing), list(thing),
               list(thing)).
:- mode concat(in, in, out) is det.

% Implementation section.
:- implementation.

:- type thing ---> number(int).

thingy(N, number(N)).
concat(L1, L2, L3):- append(L1, L2, L3).
```

Figure 3.1: A module `app`.

we could write `call(Prepend,[4,2],Output)`. With `LL` bound to list `[3,4]`, upon completing this call, `Output` will be bound to the list `[3,4,4,2]`.

A special form of higher-order programming is the possibility of defining and using so called *type classes*. Type classes were first introduced in the context of Haskell (Wadler and Blott 1989; Hall, Hammond, Jones, and Wadler 1996). Given their elegance and the expressive power they allow, the Mercury community quickly adapted this language construct to the context of Mercury (Jeffery 2002; Jeffery, Henderson, and Somogyi 1998).

Neither higher-order programming, nor the specialised form of type classes are explicitly dealt with in this work.

3.7 Special Features

The Mercury language provides some additional features such as the use of existentially quantified types, predicates and functions, interfaces with foreign languages (C, Java, .NET), and many syntactic sugar bits. As these aspects are not specifically dealt with in this thesis, we omit any further description here and refer the reader to the Reference Manual for further details (Henderson, Conway, Somogyi, and Jeffery 1996).

3.8 The Melbourne Mercury Compiler

As of now, there exists only one compiler and run-time system for the Mercury programming language, namely the Melbourne Mercury Compiler (MMC). Recently, this compiler has gained extra interest as it is also involved in the .NET project (Microsoft ; Platt 2003) where a common playground is created for allowing the easy interaction of programs or, on a finer level, modules and procedures, written in different programming languages.

3.8.1 Compilation Scheme

The structure of the MMC is detailed in (Conway, Henderson, and Somogyi 1995). Here we give a brief overview of some of the aspects of the compilation process that are relevant for our analyses.

The Melbourne Mercury compiler processes Mercury programs one module at a time. During the compilation of a single module, the compiler builds two intermediate internal representations of the code defined in it:

1. A high level representation of the source code, called the *High Level Data Structure*, or in short: HLDS. In this representation the source code is annotated with all relevant extra information. It also contains the information declared in the interface sections of the imported modules;

2. The second representation is called the *Low Level Data Structure*, in short LLDS, and represents the source code in terms of low level instructions that can almost directly be mapped onto statements in the back-end language of choice (in most cases these are statements in C).

A normal compilation process of a MMC-Mercury module roughly follows the following scheme:

1. First the source code is parsed, and stored in the HLDS.

2. Semantic analyses, error checking and high-level transformations use that HLDS as input, and update this structure.

3. Next, the code generation pass transforms the high level source code information recorded in the HLDS into the low level representation of the LLDS.

4. The LLDS can undergo a number of low-level optimisation passes and finally, the target code is produced.

The Melbourne Mercury compiler can produce target code of different types, called different *back-ends*. The most popular back-ends are low-level C code (close to assembler) (Somogyi, Henderson, and Conway 1996), high-level C code (Henderson and Somogyi 2002), or nowadays also .NET (Dowd, Henderson, and Ross 2001).

The analyses that we describe in this thesis are all analyses that fit into the second step of the described compilation process. This allows us to still manipulate (almost) source code, yet the code is already checked and transformed to suit our needs:

- For each mode declared for a predicate, a separate procedure is generated;
- The procedures are already verified w.r.t. mode, type and determinism information. As a consequence of the mode verification, the compiler may have reordered the goals within a procedure in such a way that all variables always get instantiated before they are used;
- All unifications are specialised into either constructions, deconstructions, tests or assignments;
- Every goal and literal is explicitly annotated with mode and determinism information, and also the type of every involved variable can easily be queried.

The purpose of our analysis-system is to annotate the code with low-level instructions that reuse data that has become available for reuse. This means that the order between the literals producing the available data, and the literals reusing that data need to be guaranteed. Hence, we must ensure that our analysis system is not followed by any compiler pass that can possibly reorder the literals within a procedure definition.

3.8.2 Interface Files

As already sketched above, the Melbourne Mercury compiler compiles each module of a program one at a time. Yet, during the compilation of one module, the compiler also needs some part of the information present in the modules imported by that module such as the procedure declarations of the exported procedures to verify the mode-, type- and determinism-correctness of the module being

compiled. To avoid the need of fully loading the code of the imported modules, the Melbourne Mercury compiler makes use of *interface* files. The interface file of a module records all the information about that module that is necessary for the correct compilation of the modules importing that module. Such an interface file mainly contains the type, mode and procedure declarations of the exported entities.

Additional interface files are used for extra analysis information. For example, the Mercury compiler is able to do some termination checking (Speirs, Somogyi, and Søndergaard 1997). For this purpose, the results of analysing a module are recorded in an *optimisation interface* file. When analysing other modules, the termination results of already analysed modules can therefore simply be accessed by looking at these files.

For our analyses we also make use of optimisation interface files for recording the analysis results of the modules.

3.8.3 Term Representation

The purpose of the CTGC system is to identify which objects on the heap, so called *data structures*, become unused at a certain moment during the execution of a program. In order to understand what these objects are, we clarify the way typed terms are usually represented in the Melbourne Mercury Compiler. Note that the Melbourne Mercury Compiler compiles to different back-ends, the most common being ANSI-C. Higher-level back-ends, such as Java or .NET, use different low level representations, yet the basic concepts remain the same.

Consider the following types:

```
:- type dir ---> north ; south ; east ; west.
:- type example ---> a(int, dir) ; b(example).
```

Terms of primitive types such as integers, chars, floats (or pointers to floats, depending on the word-size of the underlying machine) and pointers to strings are represented as single machine words. Terms of types such as dir, in which every alternative is a constant, are represented in the same way as enumerated types in other languages. Mercury represents them as consecutive integers starting from zero, and stores them in a single machine word. Terms of types such as example are stored on the heap. Unlike in usual logic program implementations, the function symbols of the stored terms are not explicitly recorded together with these stored terms, and tags can be used instead. Indeed, as Mercury is a strictly typed language, all types are known to the compiler. This allows the use of simple tags to identify the functors of terms of a given type (Somogyi, Henderson, and Conway 1996; Dowd, Somogyi, Henderson, Conway, and Jeffery 1999). These tags are stored in the two lowest-order bits of the memory word in which the pointer to the term is stored. The result is a space aware and more importantly time efficient implementation.

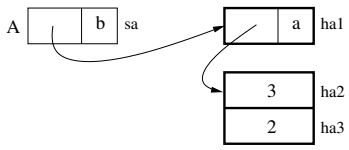


Figure 3.2: A = b(a(3,east)).

For those types which have more functors than a simple tag, also called *primary* tag can distinguish, a secondary tag is used. In some cases, no extra memory word is required to store the secondary tag (using one of the non-lowest-order bits of the memory word), yet in general, secondary tags are recorded as extra words in the memory block representing the stored term.

As the exact use of primary or secondary tag is irrelevant for the theory of the present exposition, we assume that all terms can be represented using primary tags only.

Figure 3.2 shows the representation of a variable A bound to a term b(a(3,east)) of type example defined above. In the following figures, ha1, hy1,... denote heap cells, whereas sa, sx,... are registers or stack locations.

3.8.4 Run-Time Garbage Collector

Consider the definition of a procedure convert(A,B) that converts terms of type example to new terms of that type as given by the code in Figure 3.3

```
%:- pred convert(example, example).
%:- mode convert(in, out) is semidet.
convert(X,Y):- X => b(X1),
               X1 => a(A1, _),
               Y1 <= a(A1, north),
               Y <= b(Y1).
```

Figure 3.3: Conversion-procedure.

Figure 3.4 shows the memory layout when calling convert(A,B) where A is bound to the term b(a(3,east)) as depicted in Figure 3.2.

After deconstructing the input, the run-time system allocates a new block of heap cells (hy1-hy3) to create the output term bound to Y where the content of X is partially copied to those cells. If the original term stored in the heap cells ha1-ha3 is not accessed anymore during the remainder of the execution of the

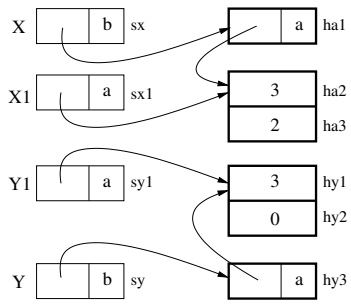


Figure 3.4: Memory layout when executing convert(A,B) with A bound to b(a(3,east)) without structure reuse.

program, then these cells are considered *garbage*. Currently, it is up to the run-time system and its run-time garbage collector, to detect and collect such garbage cells.

The MMC mainly relies on Hans Boehm's conservative garbage collector for C (Boehm and Weiser 1988). Essentially, this collector halts the execution of the program, traces the live objects on the heap, marks them, and examines and collects the potentially dead objects. This means that the time between the moment that the heap cells `ha1`-`ha3` become dead, and the moment that they are detected as dead, can be fairly large. In the presence of a compile-time garbage collection system, which is the aim of our work, we could rely on program analysis to determine that a particular call to convert leaves dead heap cells. In this example, if we can show at compile-time that after the particular procedure call of convert(A,B), the term pointed at by X will not be referenced during the rest of the program (thus becoming available for reuse), then the deconstruction statements perform the last access ever to the concerned heap cells (`ha1`, `ha2`, `ha3`) after which they become garbage. The compiler can then decide to reuse these heap cells for creating Y, in which case the time between heap cells becoming garbage and that garbage being reused for new objects can be greatly reduced. This desired situation is depicted in Figure 3.5.

3.9 Conclusion

In this chapter we presented the essential elements of the Mercury programming language, the language for which the analyses developed in this thesis are developed. As these analyses are also implemented into the Melbourne Mercury compilers, some details about the structure and run-time system of that specific compiler are presented.

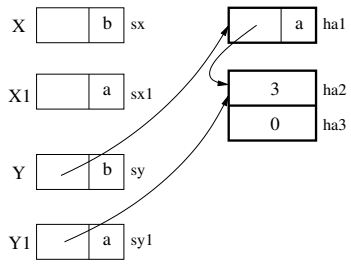


Figure 3.5: Memory layout when executing `convert(A,B)` with `A` bound to `b(a(3,east))` with structure reuse.

Chapter 4

Core Mercury Syntax

We introduce a first-order subset of Mercury. This is the target language for the next chapters. Therefore, unless explicitly stated otherwise, *Core* Mercury programs are simply referred to as Mercury programs. When needed, actual Mercury programs — *i.e.*, conforming to the language definition of (Somogyi, Henderson, and Conway 1996) — are called *plain Mercury* programs.

4.1 Language Definition

We restrict our language to first-order non-modular programs.

The formal syntax that we define for Mercury programs differs in a number of ways from the actual syntax of these programs in the sense that we assume that a number of semantics-preserving transformations have been done on the original code, hence obtaining programs conforming to our stricter syntax-rules.

These are the assumed transformations:

- All functions and function calls are replaced by predicates and predicate calls respectively. This is a natural transformation as already mentioned in Section 3.1.

- Each predicate definition is replaced by a set of procedure definitions, one procedure for each mode declared for that predicate.

- Each procedure is defined by exactly one procedure clause. As Mercury allows the use of explicit disjunctions, this effect can simply be achieved by replacing the arguments of the head of each of the procedure clauses by a same set of variables, possibly adding explicit unifications between these fresh variables and the original arguments, and finally, putting each

clause as a different branch of an explicit disjunction. This disjunction then becomes the procedure code.

- All programs are correct with respect to the declared types, modes, and determinisms. This implies that the literals in procedures may be rearranged w.r.t. the initial predicate definition.
- Every unification is specialised in one of its four forms (c.f. Section 4.2.3).
- And finally, all programs should be *normalised* in the sense that all arguments of the terms and atoms[1] appearing in our Mercury programs should be distinct variables.

These assumptions lead us to the formal syntax of the Mercury language as depicted in Figure 4.1. Note that procedures are formally written as $h \leftarrow g$, but in concrete examples we continue to use `:-` to separate the head from the goal. In analogy, in formal equations we write unifications using $:=, ==, \Leftarrow$ and $\Rightarrow$, while in concrete pieces of program code we use `:=`, `==`, `<=` and `=>` respectively.

We give a brief description of each of the elements in the language.

A Mercury program consists of a set of procedures, and a query.

In a plain Mercury program, the query is always a call to a procedure named `main/2` that needs to be implemented by the programmer. We generalise this, and assume that the query can be any kind of goal.

Each procedure is defined by a rule $p(X_1, \ldots, X_n) \leftarrow g$. The *head atom* of this definition is the atom $p(X_1, \ldots, X_n)$, and the *body* consist of a goal g. Mercury goals can either be a conjunction of goals, a disjunction of goals, a negation of a goal, an if-then-else construct, or a simple literal. Finally, a literal l can either be an explicit unification specialised to one of its four forms (See also Section 4.2.3) or a procedure call.

We introduce some extra notation:

Definition 4.1 (Subgoal) *Let g and g' be procedure goals. We say that g' is a* subgoal *of goal g iff g' occurs in goal g. This relationship is a partial order and is denoted by $g' \leq g$.*

Example 4.1 *Let $g = (g_1; g_2), (g_3; (g_4, g_5))$, then $\forall g_i, 1 \leq i \leq 5 : g_i \leq g$ but also $(g_1; g_2) \leq g, (g_4, g_5) \leq g$, etc.*

In a Mercury program the programmer needs to explicitly adorn her/his program with type-, mode- and determinism declarations. We consider that this information is implicitly present in our transformed Mercury programs. This is presented in the following sections.

Example 4.2 *The code in Example 3.9 meets the definition of the syntax of Mercury programs given here.*

[1] Note that explicit unifications are not considered as atoms in our formalism, see Section 3.1.

$$
\begin{array}{lcll}
\textit{Program} & ::= & r\,;\,q & \\
\textit{RuleBase} & ::= & \{p_1 \ldots p_{n_p}\} & n_p \geq 1 \\
\textit{Procedure} & ::= & p(\overline{X}) \leftarrow g & \\
\textit{Goal} & ::= & g_1, g_2 & \\
 & | & g_1; g_2 & \\
 & | & \text{if } g_1 \text{ then } g_2 \text{ else } g_3 & \\
 & | & \text{not } g & \\
 & | & l & \\
\textit{Literal} & ::= & X \Rightarrow f(\overline{Y}) & \text{(deconstruction)} \\
 & | & X \Leftarrow f(\overline{Y}) & \text{(construction)} \\
 & | & X == Y & \text{(test)} \\
 & | & X := Y & \text{(assignment)} \\
 & | & p(\overline{Y}) &
\end{array}
$$

where

$$
\begin{array}{lcl}
r & \in & \textit{RuleBase} \\
\{p_1, \ldots, p_{n_p}\} & \subseteq & \textit{Procedure} \\
\{q, g, g_1, g_2, g_3\} & \subseteq & \textit{Goal} \\
l & \in & \textit{Literal} \\
\{X, Y, X_1, \ldots, X_n, Y_1, \ldots, Y_m\} & \subseteq & \mathcal{V} \\
f/n & \in & \Sigma
\end{array}
$$

Figure 4.1: Description of core Mercury. Note that n_p denotes the number of procedures in the program and $X_1, \ldots, X_n$, $Y_1, \ldots, Y_m$ and $\overline{X}$ each denote a sequence of distinct variables.

4.2 Implicit Information

The following sections present the extra information that we consider to be implicitly present in our transformed Mercury programs.

4.2.1 Program Point and Execution Path

We identify each individual literal by a unique *program point*.

The set of program points is denoted by $\mathtt{pp}$; to designate the literal at some specific program point i, we use the notation l_i; the program point of a literal l is denoted as $\mathtt{pp}(l)$; the set of program points occurring in the definition of a goal g is given by $\mathtt{pp}(g)$, and finally, the set of program points in the definition of a procedure p is written as $\mathtt{pp}(p)$. If we explicitly write a program point within a fragment of code we do this by writing it in front of the literal to which it refers.

The reason for doing so will become clear when we present the notion of program point annotations and their meaning (Section 5.3.1).

The set of program points of a given procedure is ordered w.r.t. the syntactical occurrence of the literals they correspond to. As program points are usually identified by integer numbers, we use the symbol $\leq$ to denote the ordering of program points.

Example 4.3 *Program points are marked before the literals to which they belong to.*

```
%:- pred first(list(T), T).
%:- mode first(in, out) is det.
first(L,F) :- (1) L => [X | _R], (2) F := X.
```

Here we have $\texttt{pp}(\texttt{L} = [\texttt{X}|\texttt{\_R}]) = (1)$*, and* $\texttt{pp}(\texttt{F:=X}) = (2)$*;* $\texttt{pp}(\textit{ first /2}) = \{(1), (2)\}$*. If we consider that our program consists of only this procedure then* $\texttt{pp} = \texttt{pp}(\textit{ first /2}) = \{(1), (2)\}$*. In this simple example, obviously,* $(1) \leq (2)$*.*

For some of the analyses it is important to know which literals precede the execution of a given literal, and which literals can only be performed after that literal, both in the context of the same procedure. For this purpose we introduce the notion of an *execution path*, also called a *control flow path* in (Vanhoof 2001). Execution paths are statically determined by the selection rule, here left-to-right.

Definition 4.2 (Execution Path) *Given a procedure* $p(\overline{X}) \leftarrow g$*, an* execution path *in this procedure is a sequence of program points* $\langle pp_1, pp_2, \ldots, pp_n \rangle$*, where* $pp_i < pp_{i+1}, 1 \leq i < (n-1)$*, such that at run-time the literals associated with these program points are performed in sequence. A program point is said to be* covered *by an execution path, if this execution path comprises that particular program point. Two program points* share an execution path *if there exists an execution path covering both program points. Two execution paths are* sharing *if the intersection between the two corresponding sequences is not empty.*

Given a procedure p, then each execution path in its definition, is denoted by $\overrightarrow{p_i}$ where i is an index allowing to differentiate each of these paths. We can order these execution paths based on the ordering of the program points they cover, let $\overrightarrow{p_i}$, $\overrightarrow{p_j}$ be two execution paths in p, then $\overrightarrow{p_i}$ is *before* $\overrightarrow{p_j}$, denoted by $\overrightarrow{p_i} \overrightarrow{\leq} \overrightarrow{p_j}$, iff $\forall x \in \overrightarrow{p_i}, \exists y \in \overrightarrow{p_j} : x \leq y$. The set of execution paths of a procedure p is simply denoted by $\texttt{paths}(p)$.

Example 4.4 *Consider the definition of the deterministic procedure of* append/3 *(Example 3.9), here explicitly annotated with its program points:*

```
%:- pred append(list(T), list(T), list(T)).
%:- mode append(in, in, out).
append(X, Y, Z) :-
```

```
(
  (1) X => [ ] ,
  (2) Z := Y
;
  (3) X => [Xe|Xs] ,
  (4) append(Xs, Y, Zs) ,
  (5) Z <= [Xe|Zs]
) .
```

This procedure has two execution paths, namely $\overrightarrow{\texttt{append}_1} = \langle(1),(2)\rangle$*, and* $\overrightarrow{\texttt{append}_2} = \langle(3),(4),(5)\rangle$*. Each program point is only covered by one single execution path, there is no sharing between execution paths. Note that*

$$\overrightarrow{\texttt{append}_1} \overrightarrow{\leq} \overrightarrow{\texttt{append}_2}$$

In the following sections and chapters we will often refer to this version of the deterministic append procedure.

Example 4.5 *Consider the following sketch of a procedure definition:*

```
r :- (1) s ,
     ( (2) t ; (3) u ) ,
     ( (4) v ; (5) w ) .
```

Then

$$\texttt{paths}(r) = \{\langle(1),(2),(4)\rangle, \langle(1),(3),(4)\rangle, \langle(1),(2),(5)\rangle, \langle(1),(3),(5)\rangle\}$$

Here, $\langle(1),(2),(4)\rangle$ *is before any of the other paths,* $\langle(1),(3),(5)\rangle$ *is after any of the other paths, while* $\langle(1),(2),(5)\rangle$ *and* $\langle(1),(3),(4)\rangle$ *are incomparable.*

It is possible to prove that the set of execution paths of any procedure p always has a lowest element, *i.e.*, a path $\overrightarrow{p_\perp} \in \texttt{paths}(p)$ such that $\forall \overrightarrow{p_i} \in \texttt{paths}(p) : \overrightarrow{p_\perp} \overrightarrow{\leq} \overrightarrow{p_i}$.

Definition 4.3 (Preceding, Following Program Points) *Consider a procedure* $p(\overline{X}) \leftarrow g$*, and a program point* $i \in \texttt{pp}(g)$*. The program points* preceding *(resp.* following*) i is the union of the program points preceding (resp. following) i in the execution paths covering i. The set of preceding program points is denoted by* $\texttt{pre}(i)$*, while the set of following program points is given by* $\texttt{post}(i)$*. Formally:*

$$\begin{array}{lcl} \texttt{pre}(i) & = & \{j \mid \overrightarrow{p} \in \texttt{paths}(p), \{i,j\} \subseteq \overrightarrow{p}, j \leq i\} \\ \texttt{post}(i) & = & \{j \mid \overrightarrow{p} \in \texttt{paths}(p), \{i,j\} \subseteq \overrightarrow{p}, i \leq j\} \end{array}$$

Example 4.6 *In Example 4.5,* $\texttt{pre}(1) = \{\}, \texttt{post}(1) = \{(2),(3),(4),(5)\}$*, while for program point* (3) *we have* $\texttt{pre}(3) = \{(1)\}$ *and* $\texttt{post}(3) = \{(4),(5)\}$.

We generalise the definition of preceding and following program points for goals instead of individual program points.

Definition 4.4 *Given a procedure $p(\overline{X}) \leftarrow g$, and a subgoal $g' \leq g$. The program points* preceding *(resp. following) g' is the intersection of the program points preceding (resp. following) the program points $\texttt{pp}(g')$. We overload the meaning of* `pre` *and* `post`*:* $\texttt{pre}(g')$ *denotes the set of program points preceding g', and* $\texttt{post}(g')$ *is the set of program points following g'.*

4.2.2 Type Information

In the context of one procedure definition, each variable has a unique type. We introduce the function $\texttt{type}(X, p)$ which returns the type of variable X used in the definition of procedure p.

In general, if the procedure context is known, $\texttt{type}(X, p)$ is abbreviated to $\texttt{type}(X)$.

4.2.3 Mode Information

During mode analysis, the goal of a predicate is transformed in such a way that after reordering of the subgoals, the resulting code ensures that, when executed, all variables are given a value before they are used. General predicate calls are replaced by adequate procedure calls. Propagating and inferring mode information also has the consequence that for each general unification it is possible to know the flow of information. Each unification is therefore specialised to one of the following four unifications (Somogyi, Henderson, and Conway 1996):

- (*deconstruction*) A deconstruction unification $X \Rightarrow f(Y_1, \ldots, Y_n)$ is a unification in which X has mode `in`, and all $Y_i \in \{Y_1, \ldots, Y_n\}$ have mode `out`. This unification may fail if X is not bound to the outermost functor f/n. In other words, if X is bound to a ground term $f(Z_1, \ldots, Z_n)$, then the result of the unification will be that each Y_i will be bound to the subterm Z_i points to, $1 \leq i \leq n$.

- (*construction*) A construction unification $X \Leftarrow f(Y_1, \ldots, Y_n)$ is a unification in which all $Y_i \in \{Y_1, \ldots, Y_n\}$ are input, while the left hand side, X, is output. This means that a new term is constructed with outermost functor f/n and where the subterms all point to the corresponding $Y_i, 1 \leq i \leq n$. The resulting term is assigned to the fresh variable X. This type of unification can not fail.

- (*test*) A test unification $X == Y$ tests whether the ground terms pointed at by X and Y are equal. This means that both X and Y are input variables. The unification fails if the terms are not equal.

- (*assignment*) An assignment $X := Y$ simply assigns the value of the input variable Y to the fresh variable X. This type of unification can not fail.

We introduce the following query functions related to mode information and which can be applied to any syntactic object, say S, of our language:

- $\texttt{in}(S)$: returns the set of variables which have mode `in` with respect to S. These are variables which are guaranteed to be ground by the time the syntactic object S is executed.
- $\texttt{out}(S)$: returns the set of `out`-variables in S. These are variables that were free before calling S and are instantiated upon successful execution of S.
- although rarely used, we provide also query functions for the unique modes: $\texttt{di}(S)$ and $\texttt{uo}(S)$. These functions return the `di`, resp. `uo` variables.

Note that $\forall S : \texttt{in}(S), \texttt{out}(S), \texttt{di}(S), \texttt{uo}(S) \subseteq \mathit{Vars}(S)$.

Example 4.7 *Consider the following procedure definition:*

```
%:- pred second(list(T),T).
%:- mode second(in,out) is semidet.
second(X,X2):- X => [X1|Xs1], Xs1 => [X2|Xs2].
```

We have $\texttt{in}(p) = \{X\}$, $\texttt{out}(p) = \{X1, X2, Xs1, Xs2\}$, *where p designates the complete procedure. Restricted to the head atom of the procedure, we have*

$$\texttt{in}(\textit{second(X,X2)}) = \{X\}$$
$$\texttt{out}(\textit{second(X,X2)}) = \{X2\}$$

Other interesting mode-sets could be $\texttt{in}(X \Rightarrow \textit{[X1Xs]} \mid) = \{X\}$, *and* $\texttt{out}(X \Rightarrow \textit{[X1Xs]} \mid) = \{X1, Xs\}$.

4.2.4 Determinism Information

Using the determinism declarations provided by the programmer, any Mercury compiler must verify their correctness. Doing so, it has to derive determinism information for each part of the code. We assume that this determinism information can be queried using the function $\texttt{det}(S)$, where S is any syntactic object of our language. It returns one of the six determinism values known in Mercury: `failure`, `erroneous`, `det`, `semidet`, `multi`, `nondet` (Section 3.4).

In general, disjunctions have a non-deterministic flavour as each of the branches may succeed. There is a particular case though in which at most one of the branches succeeds. These are the so called *switches*. Just like for imperative languages, a switch compares an input value to a series of values (*i.e.*, possible outermost functors). Only one of these values can match, therefore only one of

the branches of the disjunction may lead to success. If the disjunction covers all possible outermost-functors that can be attributed to variables of that type, then the determinism of the switch depends on the determinism of the branches. If not all functors are covered, then there exists a situation where possibly none of the branches matches. In that case, the disjunction is at least semi-deterministic (it may still be non-deterministic if one of the branches is).

In our analyses we need to be able to differentiate deterministic selection, *i.e.*, switches, from general non-deterministic disjunctions. We therefore introduce an additional implicit function `switch`. Applied to a disjunctive goal, it returns the boolean value true if the goal is a deterministic switch, and false otherwise.

Example 4.8 *Consider the following program written in plain Mercury:*

```
:- type t ---> a; b.
:- pred p(int).
:- mode p(out) is multi.
:- mode p(in) is semidet.
:- pred q(t,t).
:- mode q(in,out) is det.
p(1).
p(2).
q(a,b).
q(b,b).
```

where int *is the built-in-type representing the set of integers (c.f. page 24).*

In Mercury this yields the procedure definitions `p1`*, the procedure corresponding to the non-deterministic (*`multi`*) mode of the predicate, and* `p2`*, the procedure representing the* `semidet` *mode.*

```
p1(X) :- ( X <= 1 ; X <= 2 ).
p2(X) :- ( X => 1 ; X => 2 ).
q(X,Y) :- ( X => a, Y <= b ; X => b, Y <= b ).
```

In `p1`*, the disjunction is a non-deterministic disjunction:* $\texttt{switch}(g_1) = \mathsf{false}$*, where* g_1 *is the disjunction goal of* `p1`*. In* `p2` *the disjunction is a switch, thus* $\texttt{switch}(g_2) = \mathsf{true}$*, where* g_2 *is the body of procedure* `p2`*. This switch enumerates some of the possible values X may unify with. The enumeration is not exhaustive, therefore the switch may fail (and is semi-deterministic).*

As for `q(X,Y)`*, it defines one clause for each possible input-value, therefore this disjunction can safely be marked as deterministic (it introduces no choice point, and all cases for the input-value are covered), hence a deterministic switch.*

Proposition 4.1 *The determinism of a procedure* $(h \leftarrow g)$ *is equal to the determinism of its goal* g*:* $\texttt{det}((h \leftarrow g)) = \texttt{det}(g)$.

Proposition 4.2 tables the determinism values of the unifications. We could also give an overview of the relationship between the determinism of a goal, and

the determinisms of its subgoals, but as this requires the introduction of the determinism values as a lattice, we omit it here. We simply assume that each goal has a pre-annotated determinism information field that at any time can be queried using the `det`-function.

Proposition 4.2 *The determinism information of the different specialised unifications are shown in Table 4.1.*

unif	`det`(unif)
$X \Rightarrow f(Y_1, \ldots, Y_n)$	`semidet`
$X \Leftarrow f(Y_1, \ldots, Y_n)$	`det`
$X == Y$	`semidet`
$X := Y$	`det`

Table 4.1: Determinism of unifications.

4.3 Simple Mercury

In the following chapter we give a meaning to Mercury programs. As negations and if-then-else constructs bring extra complications, we handle these constructs only in a later stage. Therefore we introduce the language Simple Mercury. This language has the same syntax and implicit information as the Mercury language we defined above, except that the goals can only be either a conjunction, a disjunction or a simple literal. No negations or if-then-else's are allowed.

Chapter 5

Mercury Semantics

In this chapter we define a number of different denotational semantics for the Mercury language defined in the previous chapter. These semantics are parametrised with respect to a so called *description domain*. By instantiating a particular semantics with a *concrete* domain, we obtain a definition of a *concrete semantics* for Mercury programs. One can also instantiate the semantics with an *abstract* domain. In such case one obtains an *abstract semantics*. If the abstract domain is designed such that its elements approximate values from a given concrete domain, then the abstract semantics obtained from instantiating the semantics with that abstract domain approximates the concrete semantics using that given concrete domain. This abstract semantics forms the basis for implementing the corresponding program analysis system. The correctness of that analysis system can then be guaranteed by the correct approximation of the abstract semantics w.r.t. the concrete semantics. As we want to be able to implement program analyses according to a different semantics than the concrete semantics that represents the actual execution of the program, we carefully design equivalence relations that enable us to relate these semantics.

The two most important semantics formalised in this chapter are:

- the *goal-dependent semantics* giving a concrete meaning to Mercury programs corresponding to the way such programs are executed, and
- the *goal-independent based semantics with pre-annotations*. This semantics is used to provide the correct basis for the consecutive analyses present in the compile-time garbage collection system developed in this thesis.

Other formalisations of the meaning of Mercury programs are used as intermediate steps in the proof of the safeness of the goal-independent based semantics with respect to the concrete goal-dependent semantics.

5.1 Introduction

5.1.1 Abstract Interpretation

The theory of *abstract interpretation*, first introduced in the context of imperative languages (Cousot and Cousot 1977) and now widely adapted to declarative languages (Cousot and Cousot 1992a; Bruynooghe 1991), allows to verify runtime properties of programs, without actually executing these programs. Such properties are useful for debugging, code optimisation, program transformation and program correctness proofs. Techniques of abstract interpretation find their application in type analysis (Kluźniak 1987; Van Hentenryck, Cortesi, and Le Charlier 1995), groundness analysis (Kågedal 1995; Cortesi, Filé, and Winsborough 1991; Marriott and Søndergaard 1993), sharing analysis (Jacobs and Langen 1992; Muthukumar and Hermenegildo 1989), combinations of these analyses (King 1994; Cortesi and Filé 1991; Muthukumar and Hermenegildo 1991; Bagnara, Zaffanella, and Hill 2000), and many other domains.

Usually, abstract interpretation is formulated in terms of the operational semantics of the language to interpret (Cousot and Cousot 1992a). Starting from a formalisation of the standard operational semantics, one formalises the concrete program properties of interest as elements of a certain *concrete domain*, thus obtaining the so called *collecting semantics* which is expressed as a fixpoint over that domain. The idea is then to translate the set of concrete values into some set of descriptions of these values, resulting in the *abstract domain*. The operations in the operational semantics that deal with concrete values need to be translated into *abstract operations* dealing with the approximate values instead. Hence, the concrete execution of a program is replaced by a pseudo-execution that is focused on the abstract properties of interest. The results of the pseudo-execution are correct if and only if they correctly approximate the concrete values that would have been obtained by actually executing the program. A good tutorial on this subject is (Bruynooghe and De Schreye 1988).

A typical example that perfectly illustrates these ideas is the rule of signs for the multiplication of real numbers.

Example 5.1 *Every child is taught the rule of signs: multiplying a negative number with a positive number must yield a negative number; if two negation signs occur in a multiplication, then the result is positive, etc. These rules are in fact abstract interpretations describing the concrete multiplication of numbers. Put differently, instead of playing with the concrete real values of the real numbers, we abstract each number by one of the elements in the set* $\{+, -, 0\}$*. This set suffices to describe the multiplication of real*

numbers as given by the following table.

$\times$	0	$+$	$-$
0	0	0	0
$+$	0	$+$	$-$
$-$	0	$-$	$+$

If we want to describe the rule of signs for the addition of two numbers, we need to add a so called top*-element, denoted as* $\top$*. This element describes the lack of knowledge about the exact sign of the number. For example, only* $\top$ *can correctly describe the addition of a negative number to a positive number. We obtain the following table:*

$+$	0	$+$	$-$	$\top$
0	0	$+$	$-$	$\top$
$+$	$+$	$+$	$\top$	$\top$
$-$	$-$	$\top$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

This led to the development of *abstract interpretation frameworks* (Bruynooghe 1991; Cousot and Cousot 1992b; Jones and Søndergaard 1987; Barbuti, Giacobazzi, and Levi 1993) and generic abstract machines that implement these frameworks (Janssens, Hermenegildo, Bueno, García de la Banda, and Mulkers 1992). For logic programming the framework of Bruynooghe (1991) is of particular interest. This framework was designed to mimic the left-to-right, depth-first search of Prolog, thus inherently top/down. The abstract engine therefore mimics the run-time computation rule. However, for our work, this framework is too strict to be usable for all the analyses that we intend to develop. We could try to design new frameworks, but for each of these frameworks a new correctness proof would be needed. The problem of choosing any fixed framework is that such a framework is always defined in terms of some specific operational semantics given to the language. In most cases this automatically determines the operational semantics of the analysis.

This motivates our choice for using a denotational approach instead. Semantics expressed in a denotational setting do not immediately reflect the implementation of the run-time execution or program analyses, hence, gives us a more flexible way of defining the concrete and abstract semantics of Mercury. In the following section we sketch the central ideas to this approach.

5.2 Denotational Abstract Interpretation

An abstract interpretation *framework* consists of a generic data flow algorithm with a few basic operations as parameters. A specific analysis is obtained by instantiating these parametric functions. The correctness of the obtained analysis is

guaranteed as long as the parametric functions correctly approximate the standard interpretation (*i.e.*, in the concrete domain) of these functions. The advantage of this approach is that a generic analysis engine can be reused for multiple analyses, as long as these analyses all follow the same structure, *i.e.*, have the same operational behaviour. The latter condition can be seen as a disadvantage of such frameworks. Indeed, if the operational semantics of the language changes (say, bottom/up instead of top/down), the framework must be adapted. Also, if the operational semantics of the language is say top/down, for some analyses it may be more interesting to have a bottom/up execution scheme. In both cases, a new framework must be designed and proved correct. In such a setting, the comparison of different frameworks is not trivial either. It is for these reasons that Marriott, Søndergaard, and Jones (1994) have introduced the notion of *denotational abstract interpretation* where program analysis is based on a denotational semantics of the language (Gordon 1979; Allison 1986; Nielson and Nielson 1992; Nielson and Nielson 1996), instead of the operational semantics. The idea is to express the underlying semantic equations of each framework in a common *meta-language*. By careful design of that meta-language, it is possible to prove the correctness of the operations at the level of the meta-language once and for all, which automatically proves the correctness of each framework described in that language (given, of course, that the chosen description domain and the operations depending on it are a safe approximation of the corresponding concrete domain and concrete operations).

Using this meta-language boils down to giving a denotational semantics to the programming language of interest. In the denotational approach, a program is seen as a composition of syntactical elements. The *meaning* of a program is defined as the *composition* of the meaning of the individual syntactical elements used in the program[1]. The semantics of each individual syntactical object is modelled by mathematical objects that represent the *effect* of executing this syntactical construct. This *effect*, which is expressed in terms of a particular description domain, is what we are interested in for modelling the behaviour of the logic programming language we use. Typical domains include the domain of variable substitutions (Lloyd 1987) or the domain of existentially quantified term equations (Marriott, Søndergaard, and Jones 1994; García de la Banda, Marriott, Stuckey, and Søndergaard 1998) (See also Chapter 2), the domain of positive boolean expressions (Marriott and Søndergaard 1993; Codish and Demoen 1993) (mainly used for tracing groundness information), the domain of Set-Sharing (Jacobs and Langen 1992; Hill, Bagnara, and Zaffanella 1998), etc.. In the context of compile-time garbage collection, we need to trace the effect the execution of a program has on the memory-bindings of the variables it uses. This requires an adequate concrete domain that captures the memory usage of the variables of

[1] Therefore, it is also common to call this approach *compositional*, as for example in (Falaschi, Gabbrielli, and Marriott 1993).

interest, and two abstract domains that respectively capture the information of *possible structure sharing* (Chapter 6), and *live* heap cells (Chapter 8).

We briefly sketch the key elements of (Marriott, Søndergaard, and Jones 1994).

The meta-language used in (Marriott, Søndergaard, and Jones 1994), which is based on (Nielson 1982; Nielson 1988), is a language made out of typed lambda expressions. The basic expressions of the language are: auxiliary functions, variables, function abstraction, function application, conditional, least fixed point, and the union and join operations. An important prerequisite of the theory is that all functions (including the auxiliary functions) must be elements of the space of monotonic functions. The union operation is explicitly added for the context of logic programming as it is needed to describe the notion of non-determinism. These expressions mainly act upon elements of a so called *description domain*. This domain must be a complete lattice. It can either be a domain representing concrete properties of the program, a so called *concrete domain*, or a domain that abstracts these properties, a so called *abstract domain*.

Definition 5.1 (Insertion) *(Marriott, Søndergaard, and Jones 1994) An* insertion *is a triple* (X, γ, Y) *where* X *and* Y *are complete lattices* $\langle X, \subseteq_X, \cup_X, \cap_X, \bot_X, \top_X \rangle$ *resp.* $\langle Y, \subseteq_Y, \cup_Y, \cap_Y, \bot_Y, \top_Y \rangle$*, and* $\gamma : Y \to X$ *is a monotonic co-strict function (*i.e., $\gamma(\top_Y) = \top_X$*). The function* γ *is called the* concretisation *function.*

The idea is that elements of the domain Y should *approximate* elements from X. In this theory the existence of an adjoined function, the so called *abstraction function* (Cousot and Cousot 1977), is not explicitly required.

Definition 5.2 (Safe approximation) *(García de la Banda, Marriott, Stuckey, and Søndergaard 1998) Let* (X, γ, Y) *be an insertion between* $\langle X, \subseteq_X, \cup_X, \cap_X, \bot_X, \top_X \rangle$ *and* $\langle Y, \subseteq_Y, \cup_Y, \cap_Y, \bot_Y, \top_Y \rangle$*. If* $x \in X$*, and* $y \in Y$ *then* y *is said to* safely approximate x*, which is written as* $y \propto x$*, iff* $x \subseteq_X \gamma(y)$*.*

The approximation operation is extended to function spaces as follows. Consider $f : X_1 \to \ldots \to X_n \to X$ *and* $g : Y_1 \to \ldots \to Y_n \to Y$ *with all* $(X_i, \gamma_i, Y_i), i = 1 \ldots n$ *insertions. We say that* g safely approximates f*, written as* $g \propto f$ *iff* $\forall x_1 \in X_1, \ldots, x_n \in X_n$ *and* $\forall y_1 \in Y_1, \ldots, y_n \in Y_n$*: if* $y_i \propto x_i$ *for* $i = 1 \ldots n$*, then* $g(y_1, \ldots, y_n) \propto f(x_1, \ldots, x_n)$*.*

We abbreviate the notion of safe approximation simply to approximation.

We use $\mathcal{S}em$ to denote a set of semantic functions used in the parametric denotational definition of a language and $\mathcal{S}em(X)$ its instantiation with a specific description domain X. If $Aux = \{a, b, c, \ldots\}$ denotes the set of auxiliary functions used in $\mathcal{S}em$, then we use the notation $\{a^X, b^X, c^X, \ldots\}$ to refer to the instantiated auxiliary function used in $\mathcal{S}em(X)$. We rephrase Theorem 4.4 in (Marriott, Søndergaard, and Jones 1994), the central theorem to proving the correctness of the denotational approach.

Theorem 5.1 (Safe approximating semantics) *(Theorem 4.4 in (Marriott, Søndergaard, and Jones 1994)) Let (X, γ, Y) be an insertion. Let $\mathcal{S}em(X)$ and $\mathcal{S}em(Y)$ be two instantiations of a set of semantic functions $\mathcal{S}em$ defined in terms of the meta-language described above, and more specifically in terms of a set of auxiliary functions Aux. If $\forall a \in Aux : a^Y \propto a^X$ then $\mathcal{S}em(Y)$ is a safe approximation of $\mathcal{S}em(X)$. This is denoted by $\mathcal{S}em(Y) \propto \mathcal{S}em(X)$.*

The use of the above notions and theorem will be made clear when applied to the context of Simple Mercury.

Finally, we write the names of the semantic functions in bold, e.g. **S**. Applying a semantic function to a syntactic element, say s, is written using the usual "syntactic" brackets '$[\![$' and '$]\!]$': $\mathbf{S}[\![s]\!]$. We use the classical functional notation when the semantic function is applied to some extra arguments: e.g. $\mathbf{S}[\![s]\!]abc$.

5.2.1 Goal-(in)dependent Semantics

We develop a number of different semantics for Mercury programs. The reason for this diversity is that a program can be interpreted in different ways, depending on the properties of interest. The two most typical views are the so called *goal-dependent semantics* and the *goal-independent based semantics*. Both semantics formulate the meaning of a Mercury program in the context of a particular program query. In the *goal-dependent* view, procedures are given a meaning in terms of the way that they are used given the initial query of the program . In the *goal-independent based* setting, the semantics of the program is based on the context-free meaning of the different procedures occurring in the program (and therefore regardless of the way these procedures are called if the program is actually executed)

Typically, for most logic programming language implementations, the goal-dependent semantics is used as a basis for the run-time behaviour of a program: the program is run top/down, left-to-right, following the typical SLD-resolution scheme (Lloyd 1987). This is why we refer to this semantics as the *natural* semantics. On the other hand, for program analysis a goal-independent based semantics can sometimes be more useful (Jacobs and Langen 1992; Codish, García de la Banda, Bruynooghe, and Hermenegildo 1997; García de la Banda, Marriott, Stuckey, and Søndergaard 1998). One of the main advantages of goal-independent based abstract semantics versus goal-dependent abstract semantics is the fact that the goal-independent meaning of the procedures can be computed once and for all. This information can then be used for each new query that needs to be analysed. If the analysis is computationally heavy, this approach saves a lot of computation time. Moreover, in the presence of modules, a goal-independent approach is often the only way of constructing the meaning of a given query without needing the full knowledge of the program. Indeed, the

goal-independent meaning of each of the procedures defined within a module can be computed and exported. This exported meaning constitutes the interface of that module to the analysis of other modules.

5.2.2 Mercury Semantics

Given the fact that the analyses that are needed in the context of a compile-time garbage collection system are relatively heavy, and given also the fact that Mercury programs are often split over a number of modules, we have decided to develop these analyses in a goal-independent based semantics context. Yet, in order to be useful, we must guarantee that the results that are obtained with these analyses are correct w.r.t. the actual run-time behaviour of the program. We could prove this correctness for each analysis separately, or define enough equivalence relations between the concrete goal-dependent semantics of a Mercury program, and its abstract goal-independent based semantics, such that only some specific properties of the abstract domain need to be satisfied in order for the full analysis to be correct. We decided to use the second approach.

Consider $\mathcal{S}em_M$ be the goal-dependent semantics for Mercury programs instantiated with the concrete domain $\wp(Eqn^+)$, then our goal is to develop a goal-independent based semantics $\mathcal{S}em_{M\bullet}$ such that if $\mathcal{S}em_{M\bullet}$ is used for a particular abstract domain $\mathcal{A}$, denoted by $\mathcal{S}em_{M\bullet}(\mathcal{A})$, and if $\mathcal{A}$ satisfies a number of properties, then the results of abstractly interpreting a Mercury program in $\mathcal{S}em_{M\bullet}(\mathcal{A})$ are correct w.r.t. the actual run-time behaviour of that program represented by $\mathcal{S}em_M(\wp(Eqn^+))$.

There are two possible ways to link $\mathcal{S}em_M(\wp(Eqn^+))$ with $\mathcal{S}em_{M\bullet}(\mathcal{A})$. Either we instantiate $\mathcal{S}em_M$ with the abstract domain $\mathcal{A}$, obtaining $\mathcal{S}em_M(\mathcal{A})$, and then prove that $\mathcal{S}em_{M\bullet}(\mathcal{A})$ is equivalent to or at least correctly approximates $\mathcal{S}em_M(\mathcal{A})$, or we give a concrete goal-independent based meaning to the original program from the beginning, and instantiate that semantics with the abstract domain $\mathcal{A}$. Both paths are shown in the following sketch:

$$\begin{array}{ccc} \mathcal{S}em_M(\wp(Eqn^+)) & \stackrel{?}{\Longleftrightarrow} & \mathcal{S}em_{M\bullet}(\wp(Eqn^+)) \\ \downarrow & & \downarrow \\ \mathcal{S}em_M(\mathcal{A}) & \stackrel{?}{\Longleftrightarrow} & \mathcal{S}em_{M\bullet}(\mathcal{A}) \end{array}$$

Either paths need to provide some proof of equivalence of the goal-independent based semantics with the goal-dependent semantics of the language. We have chosen to prove this in a generic way, independent of the particular concrete or abstract domain. For this purpose, we have introduced an intermediate semantics, the so called *differential semantics*. As we will see, in order to be equivalent, the domain (concrete or abstract) must satisfy a number of properties. We show that these properties hold for the concrete domain of existential equations, as well as for the other concrete domains constructed in this thesis.

5.3 *Simple* Semantics

We use Simple Mercury, the down sized language of Mercury, to introduce the notion of goal-dependent semantics. This semantics forms a transcription and adaptation of the semantics presented in (García de la Banda, Marriott, Stuckey, and Søndergaard 1998; Jacobs and Langen 1992). The main differences are a slight adaptation of the target language (using Simple Mercury instead of pure first-order Prolog), the notation used, and the explicit incorporation of program annotations within the semantics.

5.3.1 Semantics: Terminology and Notation

We are interested in the variable bindings that may occur as an effect of executing the program. Usually, only the bindings of the variables occurring in the query are returned as a result of the computation. These are the so called *computed answers* (Lloyd 1987; Jacobs and Langen 1992; Marriott, Søndergaard, and Jones 1994), and are expressed in terms of a specific description domain. These computed answers can be expressed in the domain of concrete variable substitutions or ex-equations, but they can also be abstracted as elements from some abstract domain describing these computed answers. In order to deal with recursive procedure definitions, the semantics uses a least fixed point. Such semantics are often called *collecting semantics*.

Example 5.2 *Consider the program defined by the rulebase containing the definition of the non-deterministic procedure of* append *as given in Example 3.9, and a query that consists of the conjunction of literals* `C <= [1], append(A,B,C)`*. Assuming that initially all of the variables are free, then the computed answers for this sequence of literals expressed in the concrete domain of variable substitutions is the set*

$$\{\{A/[1], B/[\,], C/[1]\}, \{A/[\,], B/[1], C/[1]\}\}$$

or, in the domain of ex-equations, here shown in their solved form:

$$\{A = [1] \wedge B = [\,] \wedge C = [1], A = [\,] \wedge B = [1] \wedge C = [1]\}$$

Given the fact that in the further sections we use $\wp(Eqn^+)$ as our concrete domain of reference, our subsequent illustrations are mainly in terms of $\wp(Eqn^+)$.

Our analyses enable us to determine optimisations at the level of individual literals, which means that we need to record the information about variable bindings for each of these literals. Therefore, our analyses not only return the *computed answers* for the given query of the program, but also the local descriptions of the variable bindings as they can occur during the execution of the program at the level of each single literal.

Example 5.3 *The collecting semantics for the program used in Example 5.2 results in the following* annotation *of the individual literals (also annotated with their program points) in the non-deterministic procedure of* append*:*

Literal	pp	*Annotation*
X $\Leftarrow$ []	1	$\{Z = [1], Z = [\,]\}$
Y := Z	2	$\{Z = [1] \wedge X = [\,], Z = [\,] \wedge X = [\,]\}$
Z $\Rightarrow$ [Xe|Zs]	3	$\{Z = [1], Z = [\,]\}$
append(Xs, Y, Zs)	4	$\{Z = [1] \wedge Z = [Xe|Zs]\}$
X $\Leftarrow$ [Xe|Xs]	5	$\{Z = [1] \wedge Z = [Xe|Zs]$ $\wedge Xs = [\,] \wedge Y = Zs\}$

In the previous example, we have collected the concrete variable substitutions as they may occur at run-time *before* the actual literal is taken into consideration. The other possibility would have been to collect the concrete variable bindings *after* the literal is executed. Given the fact that in this thesis we are interested in optimisations that can only be performed if literals are *called* in the right way it is only natural that we are only interested in the substitutions as they occur before the actual calls. This also explains why we have chosen to mark the program point values *before* the literal to which they belong.

Nevertheless, the *collecting semantics* presented under this form is not precise enough for our purposes. In the table in the previous example, the equations corresponding to a call of append(X,Y,Z) with Z bound to $[1]$, are mixed with the equations corresponding to calls to append in which Z is bound to the empty list. By separating these two calls we have a better view on what happens locally, making the analyses describing this information inherently more precise. Therefore, instead of annotating individual literals with the description of the variable bindings as they may occur for each call to the procedure to which the literals belong, we separate the annotations with respect to the description of the calls of the procedure. We obtain *goal-dependent annotations* of the individual literals.

Example 5.4 *The* goal-dependent annotations *for the program used in the previous example are given by the following table. Note that we have chosen to identify the literals by their program points.*

Call	pp	*Annotation*
$\{Z = [1]\}$	1	$\{Z = [1]\}$
	2	$\{Z = [1] \wedge X = [\,]\}$
	3	$\{Z = [1]\}$
	4	$\{Z = [1] \wedge Z = [Xe\|Zs]\}$
	5	$\{Z = [1] \wedge Z = [Xe\|Zs] \wedge Xs = [\,] \wedge Y = Zs\}$
$\{Z = [\,]\}$	1	$\{Z = [\,]\}$
	2	$\{Z = [\,] \wedge X = [\,]\}$
	3	$\{Z = [\,]\}$
	4	$\{\,\}$
	5	$\{\,\}$

The variable substitutions are collected w.r.t. to the way append(X,Y,Z) *is called. Note that the collected information is more precise in this case: it states that the second clause of the non-deterministic* append *fails if called with Z bound to the empty list: the literal is annotated with the bottom element* $\{\,\}$*, which reflects failure in this concrete domain.*

In the following section we give a goal-dependent semantics of Simple Mercury programs in the sense that the meaning of a program composed of a rule-base and a query q consists of the computed answer to that query and a goal-dependent annotation table that collects the relevant information for each literal. Computed answers as well as the information recorded for each literal are expressed in a description domain *Ans*. This domain is either a domain that reflects the concrete execution of the program, or an abstract domain that describes this execution.

Let *Ans* be a particular description domain, then elements of *Ans* are called *descriptions*, or sometimes *substitutions* or *patterns*. In the context of abstract interpretation or program analysis in general it is natural to use complete lattices for the description domains used which is what we also assume in our thesis. Let $\langle Ans, \sqsubseteq_{Ans}, \sqcup_{Ans}, \sqcap_{Ans}, \bot_{Ans}, \top_{Ans}\rangle$ be our description domain[2]. If a pattern describes how a specific procedure is called, then we call it a *call description*, *call substitution*, or *call pattern*. Descriptions that describe the variable bindings of a procedure after execution of that procedure are said to be *exit descriptions*, or *answer descriptions*. The element of the description domain that corresponds to the situation where all variables of interest are unbound is referred to as the *empty description*. In $\wp(Eqn^+)$, this corresponds to the value $\{\mathsf{true}\}$, in $\wp(ESubst)$, this is the element $\{\{\,\}\}$. The empty description may in some domains correspond to the bottom element $\bot$, but it is not a requirement. Using this terminology, the computed answer to a query corresponds to the answer description for that query if initially called with an empty call description.

[2]In the following sections we drop the subscripts *Ans* unless they are required for clarity.

Example 5.5 *The bottom element* $\{\}$ *in* $\wp(Eqn^+)$ *reflects failure, and is therefore different from* $\{\mathsf{true}\}$ *which represents the fact that all variables (within the context one considers) are free.*

Definition 5.3 (Goal-Dependent Annotation Table) *A* goal-dependent annotation table *is a table containing tuples from the following set:*

$$\mathtt{pp} \rightarrow Ans \rightarrow Ans$$

The intended interpretation of an element $(i, \mathcal{S}_c, \mathcal{S})$ of such a table is: if a procedure p to which the program point i belongs is called with a call description $\mathcal{S}_c$, then the particular literal at program point i is called with the description $\mathcal{S}$.

In the following sections we use the notion of updating a goal-dependent annotation table for which we introduce the following notation:

Definition 5.4 *Let T be a mapping from K-values to E-values: $T : K \rightarrow E$, where E is a complete lattice $\langle E, \sqsubseteq_E, \sqcup_E, \sqcap_E, \bot_E, \top_E \rangle$. Let $k \in K$, and $e \in E$ then*

$$T[k,e] \quad = \quad \begin{cases} T \setminus \{(k,e')\} \cup \{(k, e \sqcup_E e')\} & \text{if } \exists e' \,.\, (k,e') \in T \\ T \cup \{(k,e)\} & \text{otherwise} \end{cases}$$

5.3.2 Goal-Dependent Semantics $\mathcal{S}em_S$

This section is the first occurrence of the denotational notation we use in this text. Therefore we construct the semantics for Simple Mercury programs in a step by step way.

The goal of denotational semantics is to define mathematical functions that give a meaning to the syntactical objects used in the language. This meaning is *compositional*: the meaning of each syntactical object is expressed as a composition of the meaning of the immediate constituents of that syntactical object. In the case of Simple programs, this means that the meaning of a program $r; q$ is defined in terms of the meaning of the rulebase constituent r and in terms of the meaning of the query constituent q. If $\mathbf{P}_S, \mathbf{R}_S, \mathbf{G}_S$ are the mathematical functions giving a meaning to programs, respectively rulebases and goals, then $\mathbf{P}_S$ must be defined in terms of $\mathbf{R}_S$ and $\mathbf{G}_S$. Likewise for the meaning of the other syntactical objects in Simple Mercury.

Program Semantics We start by defining the mathematical function $\mathbf{P}_S$. In the previous section we announced that the meaning of a program $r; q$ is intended to be composed out of two parts: the exit description describing the variable bindings in q after completing q in the context of the rulebase r, and a goal-dependent annotation table that describes what happens at each program point in r during the execution of q. By introducing the type *Ann* as a shorthand notation

of the functions with signature $\mathtt{pp} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans}$, *i.e.*, the signature of goal-dependent annotation tables, we formalise the signature of $\mathbf{P}_S$ as being a function $\mathit{Program} \rightarrow (\mathit{Ans} \times \mathit{Ann})$ mapping individual programs to tuples containing an element from the description domain (the exit description of the query) and a goal-dependent annotation table. We define $\mathbf{P}_S$ by the clause

$$\mathbf{P}_S[\![r; q]\!] = \mathbf{G}_S[\![q]\!](\mathbf{R}_S[\![r]\!])\ \mathsf{init}_q\ \mathsf{init}_q$$

This formalises the fact that the meaning of a program $r; q$ is defined as the meaning of the goal q in the context of the meaning of the rulebase r, which is given by $\mathbf{R}_S[\![r]\!]$. The use of the extra arguments init_q will become clear in the following paragraphs. Note the use of the brackets $[\![\ ]\!]$ to clearly differentiate the syntactical objects from the mathematical objects in the semantic function definitions.

If the meaning of the rulebase is also considered part of the meaning of a particular query, then it suffices to adapt the definition of $\mathbf{P}_S$ as follows:

$$\begin{array}{rcl} \mathbf{P}_S[\![r; q]\!] & = & \mathsf{let}\ e = \mathbf{R}_S[\![r]\!]\ \mathsf{in} \\ & & \mathsf{let}\ (\mathcal{S}, A) = \mathbf{G}_S[\![q]\!]\ e\ \mathsf{init}_q\ \mathsf{init}_q\ \mathsf{in} \\ & & \quad (\mathcal{S}, A, e) \end{array}$$

The only difference of this formulation with the previous one is that now the rulebase meaning, here named e, is explicitly kept and given as a result of the meaning of the particular query q in the rulebase r. In the following chapters we shall only use the first formulation, even if we sometimes do talk about the rulebase meaning as the result of the analysis of a program.

Signature of the Rulebase Semantics In a goal-dependent context the meaning of a rulebase consists of the description of the call and exit patterns of each of the clauses, as well as the annotations of each of the program points within these clauses. The meaning of an individual procedure can then be seen as a function with the signature: $\mathit{Atom} \times \mathit{Ans} \rightarrow \mathit{Ans}$. Let $((p(X_1, \ldots, X_n), \mathcal{S}_0), \mathcal{S})$ be an element from this mapping, then it is intended to have the following interpretation: a call of $p(X_1, \ldots, X_n)$ with call description $\mathcal{S}_0$ results in the exit description $\mathcal{S}$. Let *ProcMeaning* be the shorthand notation of the functions with signature $\mathit{Atom} \times \mathit{Ans} \rightarrow \mathit{Ans}$, and *AProcMeaning* the shorthand notation for procedure meanings combined with an annotation table, *i.e.*, $\mathit{AProcMeaning} : \mathit{ProcMeaning} \times \mathit{Ann}$, we declare the signature of $\mathbf{R}_S$ to be:

$$\mathbf{R}_S : \mathit{RuleBase} \rightarrow \mathit{AProcMeaning}$$

Given a rulebase, $\mathbf{R}_S$ computes the meaning of each of the procedures defined in that rulebase, computing a goal-dependent annotation table as well.

Goal Semantics The goal-dependent meaning of a procedure goal is simple to construct. Given a call description of a goal g, the semantic function returns the corresponding exit description. This is only one part of the responsibility of the semantic function for goals. As we are explicitly interested in the intermediate values of the descriptions, we also want to update the annotation table to include the descriptions for the literals in that goal. To do so, it is not sufficient to know the call description of the goal g as this goal may be a subgoal of some larger goal. In this case we need the original call description with which the procedure was called. Therefore, the signature of $\mathbf{G}_S$ is:

$$\mathbf{G}_S : \mathit{Goal} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans} \rightarrow (\mathit{Ans} \times \mathit{Ann})$$

An expression such as $\mathbf{G}_S[\![g]\!](e, A)\mathcal{S}_0\mathcal{S}$ defines the meaning of a goal g in the context of an already precomputed rulebase meaning e and annotation table A, where the procedure to which g belongs is called with the description $\mathcal{S}_0$ (needed for correctly updating the annotation table), yet the goal itself is called with description $\mathcal{S}$. The result is an answer description describing the variable bindings after the completion of g, and a new updated annotation table.

We can now explain the double occurrence of init_q in the definition of $\mathbf{P}_S$. The semantic function $\mathbf{P}_S$ is defined in terms of the meaning of the query q. This query is simply a goal, and in order to compute its meaning, two descriptions need to be given: the call description of the procedure to which the query belongs, and the call description of the goal itself. As the query q does not belong to any procedure in particular, and the goal is called simply as is, it is natural to provide an initial empty value for both descriptions. Without fully instantiating the description domain, we introduce an *auxiliary* function init that returns a correct description of the empty set of variable substitutions, the concrete situation in which the query is executed. The signature of this function is

$$\mathsf{init} : \mathit{Ans}$$

hence a constant. In some cases the value of init may correspond to the bottom element of the description domain, but in $\wp(\mathit{Eqn}^+)$ the lack of variable bindings is simply described by an empty set of constraints, thus $\{\mathsf{true}\}$ which is not the bottom element of that domain. In some exceptional cases, this initial description might be refined with respect to the variables occurring in the context of the original query. We therefore roughly subscribe the init function with the context in which it is used. In this case, we have two occurrences of init_q.

The definition of $\mathbf{G}_S$ is given by the following clauses, where e is the precomputed rulebase meaning, A is a goal-dependent annotation table, $\mathcal{S}_0$ is the call description of the procedure to which the goal belongs, and $\mathcal{S}$ is the actual pat-

tern describing the call of the body:

$$\begin{array}{lcl}
\mathbf{G}_S[\![g_1, g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_S[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S} \text{ in} \\
 & & \quad \mathbf{G}_S[\![g_2]\!](e,A_1)\mathcal{S}_0\mathcal{S}_1 \\
\mathbf{G}_S[\![g_1; g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_S[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S} \text{ in} \\
 & & \text{let } (\mathcal{S}_2, A_2) = \mathbf{G}_S[\![g_2]\!](e,A)\mathcal{S}_0\mathcal{S} \text{ in} \\
 & & \quad (\mathcal{S}_1 \sqcup \mathcal{S}_2, \mathsf{merge}(A_1, A_2)) \\
\mathbf{G}_S[\![l]\!](e,A)\mathcal{S}_0\mathcal{S} & = & (\mathbf{L}_S[\![l]\!]\, e\, \mathcal{S}, A[(\mathtt{pp}(l), \mathcal{S}_0), \mathcal{S}])
\end{array}$$

The semantics of a conjunction of goals g_1, g_2 is defined as the meaning of the second goal g_2 computed in terms of the meaning of the first goal g_1. The meaning of a disjunction of goals $g_1; g_2$ is defined in terms of the independent meaning of each of the constituent goals, each resulting in a separate exit description and a new updated annotation table. These results are then combined into a single exit description – the least upper bound of the exit descriptions of each of the branches of the disjunction, and one single annotation table. The latter is obtained by *merging* the two annotation tables. For this purpose we introduce the merge-function, which is defined as follows.

Definition 5.5 *Let A_1 and A_2 both be annotation tables in Ann, then merging these tables is defined as:*

$$\begin{array}{l}
\mathsf{merge} : \mathit{Ann} \to \mathit{Ann} \to \mathit{Ann} \\
\mathsf{merge}(A_1, A_2) = \{(i, \mathcal{S}_0, \mathcal{S})\} \textit{ where} \\
\quad \left\{ \begin{array}{ll}
\mathcal{S} = \mathcal{S}_1 \sqcup \mathcal{S}_2 & \textit{if } (i, \mathcal{S}_0, \mathcal{S}_1) \in A_1 \wedge (i, \mathcal{S}_0, \mathcal{S}_2) \in A_2 \\
\mathcal{S} = \mathcal{S}_1 & \textit{if } (i, \mathcal{S}_0, \mathcal{S}_1) \in A_1 \wedge \nexists \mathcal{S}_2.(i, \mathcal{S}_0, \mathcal{S}_2) \in A_2 \\
\mathcal{S} = \mathcal{S}_2 & \textit{if } (i, \mathcal{S}_0, \mathcal{S}_2) \in A_2 \wedge \nexists \mathcal{S}_2.(i, \mathcal{S}_0, \mathcal{S}_1) \in A_1
\end{array} \right.
\end{array}$$

Note that we could have given a more sequential semantics to the disjunctive goals by interpreting the second branch of the disjunction using the annotation table resulting from the meaning of the first branch. Given the associative nature of the least upper bound operation, the results are guaranteed to be the same as long as the annotation table is not *consulted*, *i.e.*, as long as none of the semantic functions accesses and uses any of the values stored in the annotation table. As the annotation table is used in our definition strictly to store values without actually using these values for the analysis itself, this aspect is not an issue here.

Finally, the semantics of a goal consisting of a simple literal is defined as the semantics of that literal.

Literal Semantics The signature of the semantic function $\mathbf{L}_S$ is simply:

$$\mathit{Literal} \to \mathit{ProcMeaning} \to \mathit{Ans} \to \mathit{Ans}$$

The purpose of an expression such as $\mathbf{L}_S[\![l]\!]\, e\, \mathcal{S}$ is to return the exit description of a call of the literal l described by $\mathcal{S}$. This behaviour is defined by the following

clauses:

$$\begin{array}{lcl} \mathbf{L}_S[\![\text{unif}]\!]\, e\, \mathcal{S} & = & \mathsf{add}(\text{unif}, \mathcal{S}) \\ \mathbf{L}_S[\![p(\overline{X})]\!]\, e\, \mathcal{S} & = & e(p(\overline{X}), \mathcal{S}) \end{array}$$

where "unif" is one of the four Mercury unifications.

In this definition, we make the distinction between each of the possible forms a literal may have. If the literal is a unification, then we update the call substitution in an *appropriate way*. Given the fact that we do not yet determine the exact description domain, we introduce a monotonic *auxiliary function* add. The purpose of this auxiliary function is to update a specific description to correctly render unification[3]. Therefore, the signature of add is:

$$\mathit{Prim} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans}$$

where *Prim* represents the set of all built-in operations, *i.e.*, in this context the four different types of unification known in Mercury. The exact definition of the add function depends on the description domain that is used.
If the literal is a procedure call $p(\overline{X})$, then we *consult* the precomputed meaning of the rulebase. Given the fact that each procedure call is defined by exactly one procedure, we simply need to look up the meaning of this procedure in the precomputed rulebase meaning given the call description $\mathcal{S}$. We assume that e has been fully precomputed[4], and contains all the needed call/exit descriptions.

Rulebase Semantics We now tackle the definition of the meaning of a rulebase. By the presence of possible recursive procedures, we formalise the meaning of a rulebase r as the least fixpoint of an intermediate function $\mathbf{F}_S$ with signature:

$$\mathbf{F}_S : \mathit{RuleBase} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{AProcMeaning}$$

which is equivalent to

$$\mathbf{F}_S : \mathit{RuleBase} \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann})$$

With $\mathit{ProcMeaning} = \mathit{Atom} \times \mathit{Ans} \rightarrow \mathit{Ans}$ and knowing that in these signatures $\times$ is equivalent to $\rightarrow$ (normal currying), this allows us to write:

$$\mathbf{F}_S : \mathit{RuleBase} \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow (\mathit{Atom} \times \mathit{Ans}) \rightarrow (\mathit{Ans} \times \mathit{Ann})$$

The latter explicit form of the signature makes clear that $\mathbf{F}_S$ gives a meaning to a rulebase with respect to a precomputed rulebase meaning and goal-dependent annotation table, for a specific procedure identified by an atom, and a specific

[3] Unification is the only real built-in in Mercury, but add can be generalised to any built-in operation that might be added to Mercury.

[4] In fact, e is computed in a fixpoint computation. More about this will be said in Section 5.3.6.

call description. The result is an exit description and a new annotation table. The definition of this function is formalised by the following clause:

$$\mathbf{F}_S[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e, A)(p_i(\overline{Y}), \mathcal{S}_0) = \mathbf{Pr}_S[\![p_i]\!](e, A)(p_i(\overline{Y}), \mathcal{S}_0)$$

The definition of $\mathbf{R}_S$ is expressed using the least fixpoint operator included in our meta-language (Marriott, Søndergaard, and Jones 1994)

$$\text{fix} : (T \to T) \to T$$

where T can be any function. The definition of $\mathbf{R}_S$ is:

$$\mathbf{R}_S[\![r]\!] = \text{fix}(\mathbf{F}_S[\![r]\!])$$

How this fixpoint is computed is discussed in Section 5.3.6.

Procedure Semantics Finally, we specify the meaning of an individual procedure in the presence of a specific call. The signature of $\mathbf{Pr}_S$ can be derived from the signature of $\mathbf{F}_S$, here shown in its explicit form:

$$\mathbf{Pr}_S : \mathit{Procedure} \to (\mathit{ProcMeaning} \times \mathit{Ann}) \to (\mathit{Atom} \times \mathit{Ans}) \to (\mathit{Ans} \times \mathit{Ann})$$

We define $\mathbf{Pr}_S$ using the new monotone auxiliary function

$$\mathsf{comb} : \mathit{Ans} \to \mathit{Ans} \to \mathit{Ans}$$

The purpose of $\mathsf{comb}(\mathcal{S}_a, \mathcal{S}_b)$ is to return a correct description of the result of adding a new substitution $\mathcal{S}_b$ to an already existing substitution $\mathcal{S}_a$. The implementation of this function is of course dependent on the description domain used. The definition of $\mathbf{Pr}_S$ is given by the clause

$$\begin{array}{rcl} \mathbf{Pr}_S[\![h \leftarrow g]\!](e, A)(a, \mathcal{S}) & = & \text{let } \mathcal{S}_0 = \rho_{a \to h}\,((\mathcal{S})|_a) \text{ in} \\ & & \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_S[\![g]\!](e, A)\mathcal{S}_0\mathcal{S}_0 \text{ in} \\ & & \quad (\mathsf{comb}(\mathcal{S}, \rho_{h \to a}\,((\mathcal{S}_1)|_h)), A_1) \end{array}$$

where $(\mathcal{S})|_t$ (with t a term) is a shorthand notation for $(\mathcal{S})|_{\mathit{Vars}(t)}$, the usual projection operation (c.f. Chapter2), and $\rho_{t_1 \to t_2}\,(\mathcal{S})$ renames $\mathcal{S}$ by replacing the variables from term t_1 by the variables of its variant, term t_2 (c.f. Chapter 2).

In the context of a given rulebase meaning and annotation table, the meaning of a procedure for a specific call is defined as the meaning of the procedure goal w.r.t. a renamed and projected call description $\mathcal{S}_0$. The exit description of the goal is then renamed and combined with the original initial call description so as to bring the result back to the context of the original call.

$$
\begin{array}{lcl}
\mathit{Ans} & = & \text{description domain of interest} \\
\mathit{Ann} & = & \mathsf{pp} \times \mathit{Ans} \to \mathit{Ans} \\
\mathit{ProcMeaning} & = & \mathit{Atom} \times \mathit{Ans} \to \mathit{Ans} \\
\mathit{AProcMeaning} & = & \mathit{ProcMeaning} \times \mathit{Ann}
\end{array}
$$

Figure 5.1: Definitions of the types used in $\mathcal{S}em_S$.

$$
\begin{array}{lcl}
\mathbf{P}_S & : & \mathit{Program} \to (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{R}_S & : & \mathit{RuleBase} \to \mathit{AProcMeaning} \\
\mathbf{F}_S & : & \mathit{RuleBase} \to \mathit{AProcMeaning} \to \mathit{AProcMeaning} \\
\mathbf{Pr}_S & : & \mathit{Procedure} \to \mathit{AProcMeaning} \to \mathit{AProcMeaning} \\
\mathbf{G}_S & : & \mathit{Goal} \to \mathit{AProcMeaning} \to \mathit{Ans} \to \mathit{Ans} \to \\
 & & \quad (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{L}_S & : & \mathit{Literal} \to \mathit{ProcMeaning} \to \mathit{Ans} \to \mathit{Ans}
\end{array}
$$

Figure 5.2: Type signatures of the semantic functions used in $\mathcal{S}em_S$.

Putting it all together An overview of the semantic function definitions is presented in Figure 5.3. The types and signatures used for these definitions are recapitulated in Figure 5.1 and Figure 5.2 respectively. The semantic functions are parameterised with respect to a description domain *Ans*.

We denote the resulting semantics by $\mathcal{S}em_S$. It is parametric with respect to the exact description domain used. For each instantiation of the semantics, the auxiliary functions init, comb and add need to be defined. When instantiating the above semantics with a particular domain, say X, we denote the instantiated auxiliary functions by subscribing them with the name of the domain used: init^X, comb^X, and add^X.

The semantics is *goal-dependent* in the sense that procedures are given a meaning in the context of a particular call description. The annotations are in the same sense goal-dependent.

In the following subsection we illustrate the use of the semantic functions by instantiating it with a specific concrete domain, namely the domain of exequations. We discuss the precision of the semantic functions with respect to the real concrete execution of Mercury programs. We also present the background theory for showing that the presented semantic functions are well defined when the auxiliary functions are monotone in their description domain arguments. And finally, we discuss how these semantic functions can be implemented.

$$
\begin{array}{lcl}
\mathbf{P}_S[\![r;q]\!] & = & \mathbf{G}_S[\![q]\!](\mathbf{R}_S[\![r]\!])\mathsf{init}_q\mathsf{init}_q \\
\mathbf{R}_S[\![r]\!] & = & \mathrm{fix}(\mathbf{F}_S[\![r]\!]) \\
\mathbf{F}_S[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e,A)(p_i(\overline{Y}),\mathcal{S}_0) & & \\
 & = & \mathbf{Pr}_S[\![p_i]\!](e,A)(p_i(\overline{Y}),\mathcal{S}_0) \\
\mathbf{Pr}_S[\![h \leftarrow g]\!](e,A)(a,\mathcal{S}) & = & \mathrm{let}\,\mathcal{S}_0 = \rho_{a\to h}\,((\mathcal{S})|_a)\ \mathrm{in} \\
 & & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_S[\![g]\!](e,A)\mathcal{S}_0\mathcal{S}_0\ \mathrm{in} \\
 & & \quad (\mathsf{comb}(\mathcal{S},\rho_{h\to a}\,((\mathcal{S}_1)|_h)),A_1) \\
\mathbf{G}_S[\![g_1,g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_S[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad \mathbf{G}_S[\![g_2]\!](e,A_1)\mathcal{S}_0\mathcal{S}_1 \\
\mathbf{G}_S[\![g_1;g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_S[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \mathrm{let}\,(\mathcal{S}_2,A_2) = \mathbf{G}_S[\![g_2]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad (\mathcal{S}_1 \sqcup \mathcal{S}_2, \mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_S[\![l]\!](e,A)\mathcal{S}_0\mathcal{S} & = & (\mathbf{L}_S[\![l]\!]\,e\,\mathcal{S}, A[(\mathsf{pp}(l),\mathcal{S}_0),\mathcal{S}]) \\
\mathbf{L}_S[\![\mathrm{unif}]\!]\,e\,\mathcal{S} & = & \mathsf{add}(\mathrm{unif},\mathcal{S}) \\
\mathbf{L}_S[\![p(\overline{X})]\!]\,e\,\mathcal{S} & = & e(p(\overline{X}),\mathcal{S})
\end{array}
$$

Figure 5.3: Goal-dependent semantics $\mathcal{S}em_S$ for Simple Mercury.

5.3.3 Concrete Goal-Dependent Semantics

An interesting instantiation of the goal-dependent semantics is the concrete domain of existential equations or constraints $\wp(Eqn^+)$, a domain that we already defined in Chapter 2.

We define the *concrete goal-dependent semantics* for Simple Mercury programs as follows:

Definition 5.6 (Concrete Goal-Dependent Semantics) *Using the domain of existentially quantified ex-equations $\langle \wp(Eqn^+), \subseteq, \cup, \cap, \{\,\}, Eqn^+\rangle$ as our description domain Ans in the semantics $\mathcal{S}em_S$, we define*

$$
\begin{array}{lcl}
\mathsf{init}^{\wp(Eqn^+)} & = & \{\mathsf{true}\} \\
\mathsf{comb}^{\wp(Eqn^+)}(E,E') & = & \{e'' \mid e \in E, e' \in E', e'' = e \wedge e', e'' \textit{ is solvable}\} \\
\mathsf{add}^{\wp(Eqn^+)}(\mathit{unif},E) & = & \mathsf{comb}^{\wp(Eqn^+)}(E_{\mathit{unif}},E)
\end{array}
$$

where E_{unif} is defined as

$$
\begin{array}{lcllcl}
E_{X\Rightarrow f(\overline{Y})} & = & \{X = f(\overline{Y})\} & E_{X:=Y} & = & \{X = Y\} \\
E_{X\Leftarrow f(\overline{Y})} & = & \{X = f(\overline{Y})\} & E_{X==Y} & = & \{\,\}
\end{array}
$$

The description for $X == Y$ could also be $X = Y$, yet this equation does not represent any added value to the set of equations representing the variable bindings at the moment the test unification is considered. The resulting goal-dependent semantics is the concrete goal-dependent semantics *of our Simple Mercury language.*

pp	$\mathcal{S}_0$	$\mathcal{S}_{pp}$
1	$\{Z = [1]\}$	$\{Z = [1]\}$
2	$\{Z = [1]\}$	$\{Z = [1] \wedge X = [\,]\}$
3	$\{Z = [1]\}$	$\{Z = [1]\}$
4	$\{Z = [1]\}$	$\{Z = [1] \wedge Z = [Xe\|Zs]\}$
5	$\{Z = [1]\}$	$\{Z = [1] \wedge Z = [Xe\|Zs] \wedge Xs = [\,] \wedge Y = Zs\}$
1	$\{Z = [\,]\}$	$\{Z = [\,]\}$
2	$\{Z = [\,]\}$	$\{Z = [\,] \wedge X = [\,]\}$
3	$\{Z = [\,]\}$	$\{Z = [\,]\}$
4	$\{Z = [\,]\}$	$\{\,\}$
5	$\{Z = [\,]\}$	$\{\,\}$

Table 5.1: Goal-dependent annotation table for the non-deterministic procedure of append for the initial call description $\{Z = [1]\}$ (c.f. Example 5.6).

Note how the restriction of being solvable in the definition of $\mathsf{comb}^{\wp(Eqn^+)}$ ensures that the result is always a set of solvable ex-equations.

Example 5.6 *Consider the code of the non-deterministic procedure of* append *as in example 3.9 (here shown with its program points):*

```
append(X,Y,Z):-
        (
          (1) X <= [ ] , (2) Y := Z,
        ;
          (3) Z => [Xe|Zs],
          (4) append(Xs,Y,Zs),
          (5) X <= [Xe|Xs]
        ).
```

The concrete meaning of the query C = [1], append(A,B,C) *is the exit description:* $\{A = [\,] \wedge B = [1] \wedge C = [1], A = [1] \wedge B = [\,] \wedge C = [1]\}$ *and the goal-dependent annotation table as shown in Table 5.1. In that table* $\mathcal{S}_0$ *represents the call description of the called procedure:*

For clarity, the exit descriptions of each of the disjuncts are shown in Table 5.2, where $\mathcal{S}_0$ *again designates the call description for* append.

Obviously, append *is called with two distinct call patterns:* $\{Z = [1]\}$ *— the initial call pattern — and* $\{Z = [\,]\}$ *— the call pattern stemming from the recursive call. With the second call description, the unification of* $Z \Rightarrow [Xe|Zs]$ *fails, hence returns an unsatisfiable ex-equation, which is reflected by the* $\{\,\}$*-value for the resulting description,* i.e.*, it has no computed answer substitution. Note that the descriptions that are recorded in the annotation table are the ones that occur* before *the literal corresponding to the program point is taken into consideration. The description* after *the literal corresponding*

goal	$\mathcal{S}_0$	$\mathcal{S}$
X $\Leftarrow$ [ ], Y := Z	$\{Z = [1]\}$	$\{Z = [1] \wedge X = [\,] \wedge Y = Z\}$
Z $\Rightarrow$ [Xe|Zs], append(...), X $\Leftarrow$ [Xe|Xs]	$\{Z = [1]$	$\{Z = [1] \wedge Z = [Xe\|Zs]$ $\wedge Xs = [\,] \wedge Y = Zs \wedge X = [Xe\|Xs]\}$
X $\Leftarrow$ [ ], Y := Z	$\{Z = [\,]\}$	$\{Z = [\,] \wedge X = [\,] \wedge Y = Z\}$
Z $\Rightarrow$ [Xe|Zs], append(...), X $\Leftarrow$ [Xe|Xs]	$\{Z = [\,]$	$\{\,\}$

Table 5.2: Explicit exit descriptions for each of the disjunctions in the non-deterministic procedure of append (c.f. Example 5.6).

to a program point is not explicitly recorded, unless as the annotation of the following program point, if it exists.

These results correspond to the results detailed in Example 5.4.

5.3.4 Precision of the Concrete Semantics

The definition of the semantics for Simple Mercury was inspired by (Marriott, Søndergaard, and Jones 1994) where a similar semantics was developed for non-strongly moded definite logic programs. The authors in (Marriott, Søndergaard, and Jones 1994) mention that their definition of the concrete semantics of a clause (corresponding to our definition of a program procedure) is inherently imprecise. This is illustrated by the following fragment of Prolog code (where (1) and (2) denote the program points):

```
q(X,Y, Z) :- (1) p(X,Y), (2) r(X, Y, Z).
p(U, V) :- U = a.
p(U, V) :- V = a.
r(U,V,W) :- V = W.
```

Entering the query $\leftarrow$ q(X, Y, Z) into a Prolog system (with a left-to-right SLD-resolution scheme) yields two solutions:

$$\{X = a \wedge Y = Z, Y = a \wedge Z = a\}$$

Indeed, the call p(X, Y) either binds X or Y to a (not both), and the call r(X, Y, Z) unifies the second with the third argument.

But if we interpret this code using the semantic functions defined for Simple Mercury, then the exit description of the call p(X,Y) with call description {true}

will be the set of equations

$$\{X = a \wedge Y = Z, Y = a \wedge Z = a, \underline{X = a \wedge Y = a \wedge Z = a}\}$$

where the underlined ex-equation describes a situation that can never occur as a solution to the query. Indeed, after the evaluation of $\mathtt{p(X,Y)}$, a description $\{X = a, Y = a\}$ is obtained. This becomes the call description for $\mathtt{r(X,Y,Z)}$. Using the definition of $\mathbf{Pr}_S$, we obtain $\mathcal{S} = \{X = a, Y = a\}$, which after projecting and renaming results in $\mathcal{S}_0 = \{U = a, V = a\}$. The semantics of the procedure $r(U, V, W) \leftarrow V = W$ is to simply combine the constraint $V = W$ to any of the constraints with which the procedure is called, therefore yielding in our particular case: $\mathcal{S}_1 = \{U = a \wedge V = W, V = a \wedge V = W\}$. Finally, the resulting exit description for the procedure definition of $q(X, Y, Z)$ is obtained by applying the comb operation to the description $\{X = a, Y = a\}$ and the renamed[5] substitution $\mathcal{S}_1$:

$$\begin{array}{l} \mathsf{comb}^{\wp(Eqn^+)}(\{X = a, Y = a\}, \{X = a \wedge Y = Z, Y = a \wedge Y = Z\}) \\ = \quad \{X = a \wedge Y = Z, Y = a \wedge Z = a, \underline{X = a \wedge Y = a \wedge Z = a}\} \end{array}$$

Hence the announced imprecision. The reason why we have this imprecision comes from the fact that the original call description, a set of individual calls, is treated as a whole and therefore threaded throughout the definition of the procedure as one set. After that, the original set of individual calls are combined again with the obtained description. This way of defining the semantics does not take into account that each specific call situation results in its specific set of exit situations, but blindly combines the call description with the obtained exit description.

Therefore, if the semantic functions used for Simple Mercury are used for defining the semantics of non-strongly moded logic languages, then these functions will be inherently imprecise w.r.t. the real exit descriptions the language can produce. The reason for this loss of precision is, as shown in the previous example, the fact that, in the concrete domain, all the ex-equations describing a specific call situation, are taken together as a set, threaded as a set through the definition of the procedure goal, after which the result, again a set of equations, is unified with the original set of equations. Given the fact that in general, this unification operation is not idempotent, loss of precision can be expected.

[5] Also projected of course, but in this case this is a null operation.

A more precise description of the concrete semantics of a procedure would be given by the following clause:

$$
\begin{array}{l}
\mathbf{Pr}^{ex}_{S}[\![h \leftarrow g]\!](e, A)(a, \mathcal{S}) \\
\quad = \quad \text{let } \forall \sigma \in \mathcal{S} : \\
\qquad\quad \begin{array}{lcl}
\text{let } \mathcal{S}^{\sigma}_{0} & = & \rho_{a \to h}\left(\left(\{\sigma\}\right)\big|_{a}\right) \text{ in} \\
\text{let } (\mathcal{S}^{\sigma}_{g}, A^{\sigma}) & = & \mathbf{G}_{S}[\![g]\!](e, A)\mathcal{S}^{\sigma}_{0}\mathcal{S}^{\sigma}_{0} \text{ in} \\
\text{let } \mathcal{S}^{\sigma} & = & \mathsf{comb}\left(\{\sigma\}, \rho_{h \to a}\left(\left(\mathcal{S}^{\sigma}_{g}\right)\Big|_{h}\right)\right)
\end{array} \\
\qquad \text{in } \left(\bigcup_{\sigma \in \mathcal{S}} \mathcal{S}^{\sigma}, \mathsf{merge}_{\sigma \in \mathcal{S}} A^{\sigma}\right)
\end{array}
$$

where $\mathsf{merge}_{\sigma \in \mathcal{S}} A^{\sigma}$ is defined as the consecutive application of the merge operation on each of the intermediate annotation tables A^{σ}.

In this definition, each call description is considered to be a set of individual call descriptions (which is indeed the case for the concrete domain we use). An exit description $\mathcal{S}^{\sigma}_{g}$ of the procedure goal is computed for each individual call description $\sigma \in \mathcal{S}$. Each of these exit descriptions is then combined with the call description it stems from, resulting in a description $\mathcal{S}^{\sigma}$. Finally, all the exit descriptions are joined into one single exit description for the procedure. The individual annotation tables are merged together.

It is easy to verify that the exit description in the context of the above example would indeed correspond to the two solutions that are given by a Prolog system, namely $\{X = a \wedge Y = Z, Y = a \wedge Z = a\}$.

However, while the program code used in the above example is legal Prolog code, there is no way this code can be legal in the context of Mercury. Mercury is a strongly moded language, where each procedure is characterised by a set of input arguments, and a set of output arguments. Each clause of the initial predicate definition (or disjunction branch in a procedure) must satisfy the same mode constraints. In the definition of `p(X,Y)`, the first clause specifies X as an output argument and leaves Y unbound, while the second clause works the other way around. This behaviour can not be expressed in the Mercury mode system, hence it is not a legal Mercury program.

In fact, due to the strongly moded character of Mercury programs, the "approximating" semantics for $\mathbf{Pr}_{S}$ as given in Figure 5.3, is equivalent to the semantic function $\mathbf{Pr}^{ex}_{S}$. This property is related to the *head variable idempotence* of the $\mathsf{comb}^{\wp(Eqn^{+})}$ operation (See Item 4 of the proof of Theorem 5.5, page 88). Intuitively, if one of the variable descriptions grounds a variable, then all other descriptions must do so too to be in accordance with the modes of the procedure. Now, if an exit description stems from a particular call description, then this exit description must be modelled by that call description. Even stronger, given the fact that every single call description is different in at least one of the bindings of the concerned variable, every exit description can be modelled by exactly one of the call descriptions, namely the one it stems from. This means that only the

combination of the exit descriptions with their own call descriptions from which they stem result in solvable ex-equations. "Mixing" call descriptions with exit descriptions therefore can not happen in Mercury due to the very strict moding rules.

5.3.5 Well Definedness

We show that the goal-dependent semantics presented above exists in the sense that the fixpoint used in the definition of $\mathcal{S}em_S$ exists. The only requirement is that the auxiliary operations used in $\mathcal{S}em_S$ need to be monotonic. In a second step we show that if these auxiliary functions are also continuous, then the fixpoint process can be computed as a Kleene-sequence.

Theorem 5.2 *The natural semantics $\mathcal{S}em_S$ exists.*

Proof The auxiliary functions are monotonic over the description domain which was required to be a complete lattice. Hence the semantic function $\mathbf{F}_S$ is monotonic too. Therefore, by application of the weak form (Lloyd 1987) of Tarski's fixpoint theorem (Tarski 1955) we know that a fixpoint exists.

□

Theorem 5.3 *Consider the description domain to be a complete lattice. If that domain is finite, or if every ascending chain in the description domain used in $\mathcal{S}em_S$ is finite or if the auxiliary functions are continuous, then the semantics of the rulebase can be computed by the consecutive application of $\mathbf{F}_S$: $\mathbf{R}_S = \mathbf{F}_S^n(\bot, \bot_A)$, called the Kleene sequence, where $\bot$ is the bottom element of the lattice used for the description domain, and where $\bot_A = \{(i, \bot, \bot) \mid i \in \mathtt{pp}\}$ represents the empty annotation table.*

Proof Ascending chains are always finite in finite domains. Complete lattices in which every ascending chain is finite are called Noetherian. Noetherian lattices have the characteristic that all monotonic functions defined over these lattices are also continuous, page 401 in (Nielson, Nielson, and Hankin 1999). Hence, the auxiliary functions are continuous, which means that also $\mathbf{F}_S$ is continuous. The theorem holds by application of Proposition 5.4 in (Lloyd 1987).

□

Note that the auxiliary functions used in the concrete goal-dependent semantics $\mathcal{S}em_S(\wp(Eqn^+))$ are continuous, hence monotone ($\wp(Eqn^+)$ is a Noetherian lattice). The concrete goal-dependent semantics is therefore well defined, and the fixpoint can be computed by Kleene's sequence.

5.3.6 Possible Implementation

In the definition of $\mathbf{L}_S$ we assumed that (e, A) is fully known and that therefore a lookup operation $e(p(\overline{X}), \mathcal{S})$ always yields the exact exit description for the call description $\mathcal{S}$ of $p(\overline{X})$. This view implies a bottom-up implementation of the semantics. When evaluating the query, the rulebase meaning and annotation table corresponding to this rulebase meaning, is considered to be known. Also, when evaluating a literal, the entry for the procedure corresponding to the definition of the procedure call is considered to be present in the rulebase meaning used. Given the fact that the semantics presented here is goal-dependent, this would mean that the meaning of the rulebase would need to be computed for all possible call descriptions of the procedures defined in the rulebase, although most of these call descriptions would not be needed for the evaluation of the query of interest. This approach could be seen as a strict bottom-up approach[6]. As argued, using this approach in the implementation of the goal-dependent semantics for Simple Mercury is not feasible.

A better approach for the implementation of the goal-dependent semantics for Simple Mercury is to consider a lazy and therefore top-down oriented view. If, during the evaluation of a literal, a lookup operation is performed for a call that was not yet recorded in the rulebase meaning, then this lookup-operation triggers the evaluation of the procedure defining that procedure for the call description with which the literal was called. The rulebase meaning is therefore constructed *on demand*, hence the process becomes top-down. Note that only the calls that are needed for the evaluation of the query are evaluated in this setting.

Therefore, the most natural implementation for the goal-dependent semantics of Simple Mercury programs is a lazy top-down *on demand* oriented implementation of the fixpoint operator. This is typically the approach that can be found in some of the generic abstract interpretation frameworks developed for Prolog, such as PLAI (Muthukumar and Hermenegildo 1992), GAIA (Le Charlier and Van Hentenryck 1994) or AMAI (Janssens, Bruynooghe, and Dumortier 1995).

5.3.7 Safe Abstract Goal-Dependent Semantics

In the previous sections we have presented a parametric goal-dependent semantics for programs written in the Simple Mercury programming language. We have also defined a concrete instantiation of that semantics, namely by taking the description domain to be the domain of existentially quantified term equations $\wp(Eqn^+)$. In the following chapters we are interested in relating an *abstract* instan-

[6] Here the word *strictness* should be seen in the sense of *strict* functional languages such as Common Lisp (Steele 1984) or Standard ML (Milner, Tofte, and Macqueen 1997), as opposed to *lazy* functional languages such as Haskell (Hudak et al. 1992). In that context, a strict function is fully evaluated as soon as it is encountered, while a lazy function will only be evaluated when some of its results are needed. Lazy functions may not need to be fully evaluated if not all of the results are used.

tiation of the semantics, to a concrete instantiation. In general, we want to relate different instantiations. The specific relation that is of interest is to know whether the semantics with some instantiation $\mathcal{A}$ is a *safe approximation* of the semantics instantiated with a description domain $\mathcal{C}$. This relation of safeness is given by Theorem 5.1 (page 54). We specialise this theorem to our context of the Simple Mercury semantic functions $\mathcal{S}em_S$ and its auxiliary functions $Aux = \{\mathsf{init}, \mathsf{comb}, \mathsf{add}\}$.

Corollary 5.1 *Let $\mathcal{A}$ be a complete lattice, such that $(\mathcal{C}, \gamma, \mathcal{A})$ forms an insertion, and where $\wp(Eqn^+)$ is the concrete domain used in Definition 5.6. Let $\mathsf{init}^{\mathcal{A}}$, $\mathsf{comb}^{\mathcal{A}}$ and $\mathsf{add}^{\mathcal{A}}$ be an instantiation of the auxiliary functions used in $\mathcal{S}em_S$. If $\mathsf{init}^{\mathcal{A}} \propto \mathsf{init}^{\wp(Eqn^+)}$, $\mathsf{comb}^{\mathcal{A}} \propto \mathsf{comb}^{\wp(Eqn^+)}$ and $\mathsf{add}^{\mathcal{A}} \propto \mathsf{add}^{\wp(Eqn^+)}$ then the abstract semantics $\mathcal{S}em_S(\mathcal{A})$ is a safe approximation for the concrete semantics $\mathcal{S}em_S(\wp(Eqn^+))$,* i.e., *$\mathcal{S}em_S(\mathcal{A}) \propto \mathcal{S}em_S(\wp(Eqn^+))$.*

Example 5.7 *For this example we plan to deduce groundness information[7] and represent it using the domain of positive boolean equations known as Pos (or Prop). For more details on this domain, we refer the reader to Marriott and Søndergaard 1993; Codish and Demoen 1993; Le Charlier and Van Hentenryck 1993. It is common to extend this domain with the (non-positive) boolean element* false*. This element is used to represent the empty set of constraints (in the concrete domain). The extended domain is named $Pos_\perp$. In this domain truth-values related to variables are never capitalised. We follow that same convention here: if X is a variable, then x is used to characterise its groundness state.*

$Pos_\perp$ is a complete lattice ordered by logical consequence $\models$. The conjunction $\wedge$ and disjunction $\vee$ operations serve as greatest lower bound, resp. least upper bound. The bottom element is false*, while the top element is the boolean element* true*.*

Let θ be a variable substitution (c.f. Chapter 2), then groundθ *is a mapping of variables to truth assignments: if θ grounds a variable then it is mapped to* true*, and to* false *otherwise. Thus:* ground$\theta V \Leftrightarrow Vars(V\theta) = \{\}$*, where $V \in \mathcal{V}$. Using this notion, we define the concretisation function. We have $\gamma^{Pos} : Pos \rightarrow \wp(Eqn^+)$ where*

$$\gamma^{Pos}(\phi) \quad = \quad \{e \mid \forall \theta \in \mathsf{unif}(e) \,.\, (\mathsf{ground}\theta) \models \phi\}$$

This concretisation function is monotonic and co-strict.

Let $\mathsf{init}^{Pos_\perp}$, $\mathsf{comb}^{Pos_\perp}$ and $\mathsf{add}^{Pos_\perp}$ be defined as follows:

$$\begin{array}{lcl} \mathsf{init}^{Pos_\perp} & = & \mathsf{true} \\ \mathsf{comb}^{Pos}(\mathcal{S}_1, \mathcal{S}_2) & = & \mathcal{S}_1 \wedge \mathcal{S}_2 \\ \mathsf{add}^{Pos_\perp}(unif, \mathcal{S}) & = & \mathcal{S}_{unif} \wedge \mathcal{S} \end{array}$$

[7] Note that this information is in fact already present through the mode information available in our Mercury programs. Also, as Mercury does not allow partially instantiated data structures, all variables that get instantiated in a procedure always become ground. For this example we make abstraction of these facts.

where $\mathcal{S}_{unif}$ is defined by the following clauses:

$$\begin{array}{lcl} \mathcal{S}_{X \Leftarrow f(Y_1,\ldots,Y_n)} & = & x \leftrightarrow (y_1 \wedge y_2 \wedge \ldots \wedge y_n) \\ \mathcal{S}_{X \Rightarrow f(Y_1,\ldots,Y_n)} & = & x \leftrightarrow (y_1 \wedge y_2 \wedge \ldots \wedge y_n) \\ \mathcal{S}_{X:=Y} & = & x \leftrightarrow y \\ \mathcal{S}_{X==Y} & = & \mathsf{true} \end{array}$$

The description for $X == Y$ could also be $x \wedge y$ as both variables are definitely ground after the test, yet they must already been ground before the test, hence, the extra equation does not add any new information.

We need to prove that $\mathsf{init}^{Pos_\perp} \propto \mathsf{init}^{\wp(Eqn^+)}$, $\mathsf{comb}^{Pos_\perp} \propto \mathsf{comb}^{\wp}(Eqn^+)$ *and* $\mathsf{add}^{Pos_\perp} \propto \mathsf{add}^{\wp(Eqn^+)}$.

- $\gamma(\mathsf{init}^{Pos_\perp}) = \gamma(\mathsf{true}) = \wp(Eqn^+)$. *And* $\mathsf{init}^{\wp(Eqn^+)} = \{\mathsf{true}\} \subseteq \wp(Eqn^+) = \mathsf{init}^{Pos_\perp}$.
- *Let $E_1, E_2 \in \wp(Eqn^+)$, and $\phi_1, \phi_2 \in Pos_\perp$, such that $E_1 \subseteq \gamma(\phi_1)$, and $E_2 \subseteq \gamma(\phi_2)$, then we need to prove that* $\mathsf{comb}^{\wp(Eqn^+)}(E_1, E_2) \subseteq \gamma(\mathsf{comb}^{Pos_\perp}(\phi_1, \phi_2))$. *As* $\mathsf{comb}^{Pos_\perp}$ *is defined as the logical conjunction, and each pair of constraints is also conjoined by* $\mathsf{comb}^{\wp(Eqn^+)}$*, the logical consequence operation is satisfied.*
- *With E_{unif} as defined in Definition 5.6, clearly $E_{unif} \subseteq \gamma(\mathcal{S}_{unif})$ for every type of unification. Therefore, as* $\mathsf{comb}^{Pos_\perp} \propto \mathsf{comb}^{\wp(Eqn^+)}$*, we also have* $\mathsf{add}^{Pos_\perp} \propto \mathsf{add}^{\wp(Eqn^+)}$.

Therefore $\mathcal{S}em_S(Pos_\perp) \propto \mathcal{S}em_S(\wp(Eqn^+))$ which means that the results of abstractly interpreting a program w.r.t. $Pos_\perp$ safely approximate the concrete run-time substitutions.

Interpreting the non-deterministic call append(A,B,C) *where* C *is bound to a list [1], which can be approximated by the boolean formula c, results in the abstract exit description $a \wedge b \wedge c$, and the annotation table:*

pp	$\mathcal{S}_0$	$\mathcal{S}_{\mathrm{pp}}$
1	z	z
2	z	$x \wedge z$
3	z	z
4	z	$z \wedge x_e \wedge z_s$
5	z	$z \wedge x_e \wedge z_s \wedge (z_s \leftrightarrow x_s \wedge y)$
		$= z \wedge z_s \wedge x_e \wedge x_s \wedge y$

5.4 Goal-Dependent Semantics $\mathcal{S}em_M$

The only difference between Mercury and Simple Mercury lies in the two additional goals: negations and if-then-else constructs. Let $\mathcal{S}em_M$ be the goal-depen-

dent semantics for Mercury, then all the semantic functions $\mathbf{P}_M$, $\mathbf{R}_M$, $\mathbf{Pr}_M$, $\mathbf{G}_M$, $\mathbf{L}_M$ used in $\mathcal{S}em_M$ correspond to the semantic functions $\mathbf{P}_S$, $\mathbf{R}_S$, $\mathbf{Pr}_S$, $\mathbf{G}_S$ and $\mathbf{L}_S$ used in $\mathcal{S}em_S$. Given the two additional goals, we only need to define two extra clauses for $\mathbf{G}_M$.

We first construct this semantics from the point of view of the concrete goal-dependent semantics (Definition 5.6). We highlight some of the problems of this semantics when used for abstract domains. These problems bring us to a new, approximate, definition of the generic semantics of these two language constructs.

Example 5.8 *Consider the following procedure:*

```
%:- pred ite(list(int),int).
%:- mode ite(out, out) is nondet.
ite(X,Y) :-
        ( (1) X <= [ ] ; (2) X <= [1] ),
        if (3) X => [ ]
        then (4) Y <= 1
        else (5) Y <= 2.
```

It is obvious that there are only two possible exit descriptions for the empty call description:

$$\{X = [\,] \wedge Y = 1, X = [1] \wedge Y = 2\}$$

Defining this behaviour for the concrete domain $\wp(Eqn^+)$ results in the following additional clause for $\mathbf{G}_M$:

$$\begin{array}{l}\mathbf{G}_M[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_0\mathcal{S} = \\ \quad \text{let } (\mathcal{S}^\sigma, A^\sigma) = \mathbf{G}'_M[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_0\sigma, \quad \forall \sigma \in \mathcal{S} \\ \quad \text{in} \\ \quad\quad (\bigcup_{\sigma\in\mathcal{S}} \mathcal{S}^\sigma, \mathsf{merge}_{\sigma\in\mathcal{S}} A^\sigma)\end{array} \tag{5.1}$$

with the additional semantic function $\mathbf{G}'_M$ defined as

$$\begin{array}{l}\mathbf{G}'_M[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_0\sigma = \\ \quad \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_M[\![g_1]\!](e,A)\mathcal{S}_0\{\sigma\} \text{ in} \\ \quad \text{if } \mathcal{S}_1 \neq \{\,\} \text{ then} \\ \quad\quad \mathbf{G}_M[\![g_2]\!](e,A_1)\mathcal{S}_0\mathcal{S}_1 \\ \quad \text{else} \\ \quad\quad \mathbf{G}_M[\![g_3]\!](e,A)\mathcal{S}_0\{\sigma\}\end{array}$$

and where $\mathsf{merge}_{\sigma\in\mathcal{S}} A^\sigma$ is as defined earlier on Page 70.

This definition of the semantics of if-then-else construct can be read as follows: if $\mathcal{S}$ is a set of ex-equations, then each ex-equation is treated individually. For each of these individual call descriptions, $\mathbf{G}'_M$ defines the exit description depending on whether the tested goal g_1 fails for this call description or not. The resulting

exit descriptions and annotation tables are then combined into one single exit description and annotation table. This behaviour is similar to the detailed definition of the semantics for procedures given in Section 5.3.4.

This definition is very specific and can only be used in the context of concrete domains which represent each possible call by an individual element $\sigma \in \mathcal{S}$. It is not possible to use this definition for abstract domains where the call description usually describes a set of concrete call substitutions by one single description. E.g. a concrete call description $\{X = [\,], X = [1]\}$ consisting of two different call situations, would typically be abstracted by one single abstract formula in a *Pos*-based groundness setting: x, saying that X is ground.

Similarly, the behaviour of negated goals can be described by:

$$\begin{array}{l} \mathbf{G}_M[\![\text{not } g]\!](e, A)\mathcal{S}_0\mathcal{S} = \\ \quad \text{let } (\mathcal{S}_i, A_i) = \mathbf{G}'_M[\![\text{not } g]\!](e, A)\mathcal{S}_0\sigma_i, \quad \forall \sigma_i \in \mathcal{S} \\ \quad \text{in} \\ \quad\quad (\mathcal{S}, \mathsf{merge}_i A_i) \end{array} \tag{5.2}$$

where $\mathbf{G}'_M[\![\text{not } g]\!]$ is defined as:

$$\begin{array}{l} \mathbf{G}'_M[\![\text{not } g]\!](e, A)\mathcal{S}_0\sigma = \\ \quad \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_M[\![g]\!](e, A)\mathcal{S}_0\{\sigma\} \text{ in} \\ \quad \text{if } \mathcal{S}_1 = \{\,\} \text{ then} \\ \quad\quad (\{\sigma\}, A_1) \\ \quad \text{else} \\ \quad\quad (\{\,\}, A) \end{array}$$

The conclusions for these semantic functions are similar to the conclusions for the if-then-else construct.

In Section 5.3.4 we could simplify the definition of the semantics of procedures arguing that Mercury is a strongly moded language, hence, both definitions, whether detailed for each concrete call substitution individually, or defined for the whole set of call substitutions, are in fact equivalent. However, this reasoning can not be applied here.

Here we have two possible options: either we accept the difference between the rules describing the concrete semantics and those describing the abstract semantics, or we agree to lift the semantics for the concrete domains to a more imprecise definition yet that can be used to express the abstract semantics of the language too. The disadvantage of the first solution is that we will then need to keep track of two formalisations of the Mercury semantics, making it more cumbersome to prove the safeness of the abstract semantics (and its variations) w.r.t. the concrete semantics of the language (instead of simply proving that the auxiliary functions are safe approximations of each other, Theorem 5.1). While

loss of precision for describing the concrete semantics may be seen as a disadvantage for the second solution, as program analysis is inherently limited to *approximate* concrete execution, we argue that the loss of precision at this level is acceptable, and therefore decide for the second approach trading precision for clean formalisation.

Given the fact that at run-time, a call substitution can either satisfy the tested goal or not, but not both, we need to refer to each single call substitution of the original call description, compute its exit substitution, and combine the result. By relaxing the definition of the if-then-else and negation construct, we can avoid this behaviour. This relaxation can be done by considering that if-then-else and negated goals are equivalent to non-deterministic disjunctions:

$$\begin{array}{lcl} \text{if } g_1 \text{ then } g_2 \text{ else } g_3 & \equiv & (g_1, g_2; g_3) \\ \text{not } g & \equiv & \text{if } g \text{ then } \mathsf{false} \text{ else } \mathsf{true} \\ & \equiv & (g, \mathsf{false}; \mathsf{true}) \end{array}$$

where false and true are procedures that always fail, or always succeed respectively[8]. In this case, reconsidering the code of $ite(X, Y)$ in Example 5.8, the exit description of $ite(X, Y)$ for a call description $\mathcal{S} = \{\,\}$ becomes

$$\{X = [\,] \wedge Y = 1, \underline{X = [\,] \wedge Y = 2}, X = [1] \wedge Y = 2\}$$

The underlined ex-equation can never result from the program, so indeed, we provide an overestimation of the possible concrete substitutions.

The semantic functions corresponding to this interpretation of the if-then-else constructs and negations become:

$$\begin{array}{l} \mathbf{G}_M[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e, A)\mathcal{S}_0\mathcal{S} = \qquad (5.3) \\ \quad \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_M[\![g_1]\!](e, A)\mathcal{S}_0\mathcal{S} \text{ in} \\ \quad \text{let } (\mathcal{S}_2, A_2) = \mathbf{G}_M[\![g_2]\!](e, A_1)\mathcal{S}_0\mathcal{S}_1 \text{ in} \\ \quad \text{let } (\mathcal{S}_3, A_3) = \mathbf{G}_M[\![g_3]\!](e, A)\mathcal{S}_0\mathcal{S} \text{ in} \\ \quad\quad (\cup(\mathcal{S}_2, \mathcal{S}_3), \mathsf{merge}(A_2, A_3)) \\ \mathbf{G}_M[\![\text{not } g]\!](e, A)\mathcal{S}_0\mathcal{S} = \qquad (5.4) \\ \quad \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_M[\![g]\!](e, A)\mathcal{S}_0\mathcal{S} \text{ in} \\ \quad\quad (\mathcal{S}, A_1) \end{array}$$

These functions treat the descriptions as a whole, which makes them applicable to concrete as well as abstract description domains. It can easily be shown

[8] We have not incorporated these literals into the language, as they can be defined easily by the other syntactical elements. E.g. false :- X = 1, X = 2. and true :- X = 1.

$$
\begin{array}{lcl}
\mathbf{P}_M[\![r;q]\!] & = & \mathbf{G}_M[\![q]\!](\mathbf{R}_M[\![r]\!])\ \mathsf{init}_q\ \mathsf{init}_q \\
\mathbf{R}_M[\![r]\!] & = & \mathrm{fix}(\mathbf{F}_M[\![r]\!]) \\
\mathbf{F}_M[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e,A)(p_i(\overline{Y}),\mathcal{S}_0) & = & \mathbf{Pr}_M[\![p_i]\!](e,A)(p_i(\overline{Y}),\mathcal{S}_0) \\
\mathbf{Pr}_M[\![h \leftarrow g]\!](e,A)(a,\mathcal{S}) & = & \mathrm{let}\,\mathcal{S}_0 = \rho_{a\to h}\,((\mathcal{S})|_a)\ \mathrm{in} \\
 & & \quad \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_M[\![g]\!](e,A)\mathcal{S}_0\mathcal{S}_0\ \mathrm{in} \\
 & & \quad\quad (\mathsf{comb}(\mathcal{S},\rho_{h\to a}\,((\mathcal{S}_1)|_h)),A_1) \\
\mathbf{G}_M[\![g_1,g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_M[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad \mathbf{G}_M[\![g_2]\!](e,A_1)\mathcal{S}_0\mathcal{S}_1 \\
\mathbf{G}_M[\![g_1;g_2]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_M[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad \mathrm{let}\,(\mathcal{S}_2,A_2) = \mathbf{G}_M[\![g_2]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad\quad (\mathcal{S}_1 \sqcup \mathcal{S}_2,\mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_M[\![\mathrm{if}\ g_1\ \mathrm{then}\ g_2\ \mathrm{else}\ g_3]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_M[\![g_1]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad \mathrm{let}\,(\mathcal{S}_2,A_2) = \mathbf{G}_M[\![g_2]\!](e,A_1)\mathcal{S}_0\mathcal{S}_1\ \mathrm{in} \\
 & & \quad \mathrm{let}\,(\mathcal{S}_3,A_3) = \mathbf{G}_M[\![g_3]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad\quad (\cup(\mathcal{S}_2,\mathcal{S}_3),\mathsf{merge}(A_2,A_3)) \\
\mathbf{G}_M[\![\mathrm{not}\ g]\!](e,A)\mathcal{S}_0\mathcal{S} & = & \mathrm{let}\,(\mathcal{S}_1,A_1) = \mathbf{G}_M[\![g]\!](e,A)\mathcal{S}_0\mathcal{S}\ \mathrm{in} \\
 & & \quad (\mathcal{S},A_1) \\
\mathbf{G}_M[\![l]\!](e,A)\mathcal{S}_0\mathcal{S} & = & (\mathbf{L}_M[\![l]\!]\,e\,\mathcal{S},A[(\mathrm{pp}(l),\mathcal{S}_0),\mathcal{S}]) \\
\mathbf{L}_M[\![\mathrm{unif}]\!]\,e\,\mathcal{S} & = & \mathsf{add}(\mathrm{unif},\mathcal{S}) \\
\mathbf{L}_M[\![p(\overline{X})]\!]\,e\,\mathcal{S} & = & e(p(\overline{X}),\mathcal{S})
\end{array}
$$

Figure 5.4: Definition of the natural semantics for Mercury, $\mathcal{S}em_M$.

that the results using the first definition are always included in the results obtained using the above definition for if-then-else and negation.

Figure 5.4 recapitulates the definition of the goal-dependent semantics of Mercury programs, $\mathcal{S}em_M$. The types and signatures are the same as for Simple Mercury (Figure 5.1 and Figure 5.2 resp.).

We define the *concrete goal-dependent semantics* of Mercury to be the above semantics $\mathcal{S}em_M$ instantiated with the concrete domain of variable substitutions $\wp(Eqn^+)$. Note that this semantics is well defined.

5.5 Towards Goal-Independent Based Semantics

In the previous sections we presented goal-dependent semantics for Simple Mercury and core Mercury. These semantics correspond to the normal intuitive reading given to logic programming languages. Another semantics, often used in the context of logic programming languages, is the *goal-independent* based semantics (Jacobs and Langen 1992; Codish, García de la Banda, Bruynooghe, and

Hermenegildo 1997; García de la Banda, Marriott, Stuckey, and Søndergaard 1998). This semantics is based on the intuition that each literal and every procedure can be given a meaning regardless of the way it is exactly called or used. Given this goal-independent meaning, it suffices to combine it with an actual call description in order to obtain the corresponding exit description. Example 5.9 illustrates this idea for append in the context of a groundness analysis based on the *Pos*-domain.

Example 5.9 *Consider the new abstract call description $a \wedge c \in Pos_{\perp}$ for a call of the non-deterministic version of* append(A,B,C) *(defined in Example 5.6 at page 67). We can perform the same goal-dependent analysis as was done before for the call description c, but we can also see that each of the contributions of the literals within the definition of* append *is independent of the call description. The following table lists each of these contributions:*

pp	$\mathcal{S}$
1	x
2	$y \leftrightarrow z$
3	$z \leftrightarrow (x_e \wedge z_s)$
4	$z_s \leftrightarrow (x_s \wedge y)$
5	$x \leftrightarrow (x_e \wedge x_s)$

Indeed, regardless of how the literal $X \Leftarrow [\,]$ is called, if the call succeeds, then the variable X will always be ground, hence the boolean formula x. The reasoning behind the other literals is similar.

After each call of append*, each of these local contributions holds, and therefore, combined, we can conclude that the greatest lower bound of each of these contributions yields the net-contribution for calls to* append*. Projected onto the variables $\{x, y, z\}$ we have $\mathcal{S}_l = (x \wedge y) \leftrightarrow z$, where the subscript l refers to the* local *component of the exit description,* i.e.*, the component inherent to* append*.*

This local component can be seen as the goal-independent *contribution to calls to* append*. Combining $\mathcal{S}_l$ with specific call descriptions yields the corresponding abstract exit descriptions. In Pos this combination is done using the greatest lower bound operation, $\wedge$:*

$\mathcal{S}_{call} \Rightarrow$	$\mathcal{S}_{exit}$
$z \Rightarrow$	$x \wedge y \wedge z$
$z \wedge x \Rightarrow$	$x \wedge y \wedge z$

In fact, any call description matching the mode declaration of the procedure (third argument must be input) always yields the same exit description, namely that all three arguments of append *will be ground. This exit description corresponds to the declared output instantiation state of the procedure.*

We generalise this idea of recording the local contributions of a procedure by giving a goal-independent meaning to procedures, and using this meaning

for computing the goal-dependent annotations. The advantage of this approach is that if a procedure is called with different call descriptions, only one fixpoint computation is needed: the one for computing its goal-independent meaning. All the call specific meanings can be derived in a straightforward way. In such cases, the corresponding implementation of the semantics requires less time and effort to execute, without losing precision (if the equivalence conditions are met, see later in this chapter). Another advantage is related to the use of modules. Procedures defined in separate modules can be given a goal-independent meaning once and for all. This meaning can then be used during the analysis of the procedures defined within other modules.

The Mercury language contains negations and if-then-else constructs. In our first attempt of formalising the semantics of these constructs (Equation 5.1 and Equation 5.2), these constructs are explicitly given an inherent goal-dependent meaning. Trying to give these constructs a goal-independent meaning based on this goal-dependent semantics is useless. If the concrete call description for the negated or tested goal is not known, then the resulting exit description can not correctly be computed. This means that the semantics based on these two equations can not be given a concrete goal-independent based semantics. Interestingly enough, by giving in on the precision of the description of the concrete domain, and using Equation 5.3 and Equation 5.4, the path towards a goal-independent meaning is made easier.

The goal-independent based semantics will be defined in Section 5.7. We plan to show that if a description domain satisfies some specific properties, then the goal-independent based semantics of a program becomes equivalent to its goal-dependent semantics. In order to clearly distinguish the properties that need to be satisfied by the description domain, we use an intermediate formulation of the goal-independent based semantics, a semantics which we call the *differential semantics*.

5.6 Differential Semantics

In this section we develop a goal-dependent semantics for Mercury in which the meaning of a procedure for a given call description is defined as the local component of the final exit description of that procedure. This local component is the part of the description that is due to the procedure. The global exit description, as defined by the goal-dependent semantics for Mercury, is derived implicitly: we consider that this exit description is computed by combining the call description with the corresponding local exit description using a special combination operator (in fact using the auxiliary function comb). The semantics is still goal-dependent as the definition of the meaning of a procedure is still directed by the call description with which it is called. Yet, by separating the local contribution from the global exit description we are already one step closer towards a goal-

independent based approach in which similar local contributions are derived, yet in a separate and goal-independent way.

By the fact that in these semantics global components of the final description are explicitly separated from the local components, we called this semantics the *differential semantics* for core Mercury, denoted by $\mathcal{S}em_{M\delta}$. This is in accordance with the terminology used in (García de la Banda, Marriott, Stuckey, and Søndergaard 1998). The semantic functions for $\mathcal{S}em_{M\delta}$ are listed in Figure 5.6. The types and signatures of these functions are similar to the ones used in $\mathcal{S}em_M$. Figure 5.5 gives an overview of the signatures of these functions.

$$
\begin{array}{lcl}
\mathbf{P}_{M\delta} & : & \mathit{Program} \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{R}_{M\delta} & : & \mathit{RuleBase} \rightarrow \mathit{AProcMeaning} \\
\mathbf{F}_{M\delta} & : & \mathit{RuleBase} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{AProcMeaning} \\
\mathbf{Pr}_{M\delta} & : & \mathit{Procedure} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{AProcMeaning} \\
\mathbf{G}_{M\delta} & : & \mathit{Goal} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans} \rightarrow \\
 & & \quad (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{L}_{M\delta} & : & \mathit{Literal} \rightarrow \mathit{ProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans}
\end{array}
$$

Figure 5.5: Type signatures of the semantic functions used in $\mathcal{S}em_{M\delta}$.

The set of auxiliary functions used in $\mathcal{S}em_{M\delta}$ is $\{\mathsf{init}, \mathsf{comb}, \mathsf{add}\}$. The merge operation used in Figure 5.6 is the same operation that was introduced in $\mathcal{S}em_S$ and $\mathcal{S}em_M$ (Definition 5.5, page 62).

The intention of the semantics $\mathcal{S}em_{M\delta}$ is to make a distinction between a *global* component and a *local* component of the descriptions that are defined. In this semantics $\mathcal{S}_g$, the *global* component, is meant to designate the part of the description that is due to the call of the procedure to which the current language construct belongs. This component remains the same for each of the language constructs occurring within the same procedure call. The description subscribed with the character l, $\mathcal{S}_l$, is meant to capture the part of the description that is accumulated through the influence of the language constructs that have preceded the current language construct within the procedure call it belongs to. This is the so called *local* component. The complete exit description, denoted by $\mathcal{S}_c$, is the combination of the original call description $\mathcal{S}_g$ and the locally computed exit description $\mathcal{S}_l$: $\mathcal{S}_c = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$. Figure 5.7 gives a sketch of the relation between these individual descriptions.

The resulting meaning of a goal or a literal is an update of that local component and the annotation table related to the procedure call. The fact that $\mathcal{S}_l$ is meant to denote the local component of the full exit substitution can also be seen by the fact that the meaning of a procedure is given in terms of the meaning of its goal w.r.t. the global component $\mathcal{S}_g$, derived from the call substitution $\mathcal{S}$, and an initial local component, initialised using the init-function. Note that the resulting

$$
\begin{array}{lcl}
\mathbf{P}_{M\delta}[\![r;q]\!] & = & \mathbf{G}_{M\delta}[\![q]\!](\mathbf{R}_{M\delta}[\![r]\!])\ \mathsf{init}_q\ \mathsf{init}_q \\
\mathbf{R}_{M\delta}[\![r]\!] & = & \mathrm{fix}(\mathbf{F}_{M\delta}[\![r]\!]) \\
\mathbf{F}_{M\delta}[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e,A)p_i(\overline{Y}) & & \\
 & = & \mathbf{Pr}_{M\delta}[\![p_i]\!](e,A)p_i(\overline{Y}) \\
\mathbf{Pr}_{M\delta}[\![h \leftarrow g]\!](e,A)(a,\mathcal{S}) & = & \mathsf{let}\ \mathcal{S}_g = \rho_{a\to h}\left((\mathcal{S})|_a\right)\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\delta}[\![g]\!](e,A)\mathcal{S}_g\mathsf{init}_{h\leftarrow g}\ \mathsf{in} \\
 & & \quad\quad (\rho_{h\to a}\left((\mathcal{S}_{l_1})|_h\right), A_1) \\
\mathbf{G}_{M\delta}[\![g_1,g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \mathsf{let}\ (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\delta}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad \mathbf{G}_{M\delta}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1} \\
\mathbf{G}_{M\delta}[\![g_1;g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \mathsf{let}\ (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\delta}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M\delta}[\![g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad\quad (\mathcal{S}_{l_1} \sqcup \mathcal{S}_{l_2}, \mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_{M\delta}[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l & & \\
 & = & \mathsf{let}\ (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\delta}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M\delta}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1}\ \mathsf{in} \\
 & & \quad \mathsf{let}\ (\mathcal{S}_{l_3}, A_3) = \mathbf{G}_{M\delta}[\![g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad\quad (\mathcal{S}_{l_2} \sqcup \mathcal{S}_{l_3}, \mathsf{merge}(A_2,A_3)) \\
\mathbf{G}_{M\delta}[\![\text{not } g]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \mathsf{let}\ (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\delta}[\![g]\!](e,A)\mathcal{S}_g\mathcal{S}_l\ \mathsf{in} \\
 & & \quad (\mathcal{S}_l, A_1) \\
\mathbf{G}_{M\delta}[\![l]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & (\mathbf{L}_{M\delta}[\![l]\!]\, e\, \mathcal{S}_g\mathcal{S}_l, \\
 & & \quad A[(\mathsf{pp}(l), \mathcal{S}_g), \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)]) \\
\mathbf{L}_{M\delta}[\![\text{unif}]\!]\, e\, \mathcal{S}_g\mathcal{S}_l & = & \mathsf{add}(\text{unif}, \mathcal{S}_l) \\
\mathbf{L}_{M\delta}[\![p(\overline{X})]\!]\, e\, \mathcal{S}_g\mathcal{S}_l & = & \mathsf{comb}(\mathcal{S}_l, e(p(\overline{X}), \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)))
\end{array}
$$

Figure 5.6: Differential semantics $\mathcal{S}em_{M\delta}$.

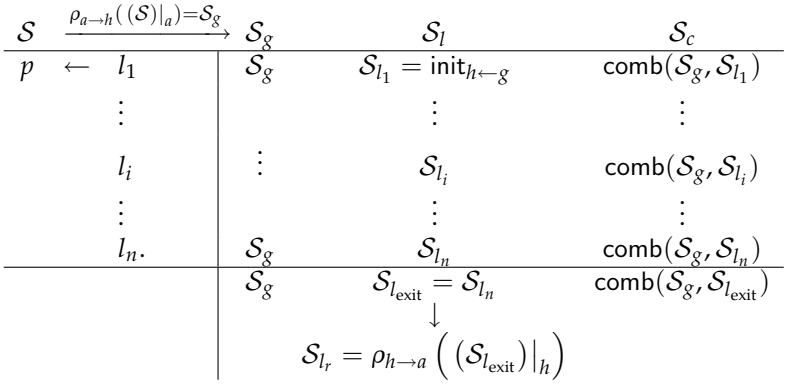


Figure 5.7: Schematic representation of the semantics of a procedure $p \leftarrow l_1, \ldots, l_n$ for a call description $\mathcal{S}$ in $\mathcal{S}em_{M\delta}$.

meaning of that goal is a renaming of the local component $\mathcal{S}_{l_1}$ projected on the head of the procedure. This projection is needed as we specifically record only the local contributions of procedure calls to their exit descriptions.

Most of the rules defining $\mathbf{G}_{M\delta}$ and $\mathbf{L}_{M\delta}$ are straightforward. In the semantics for procedure call literals, we define the local exit description to be the combination of the local part of the call description ($\mathcal{S}_l$) with the result of looking up the exit description of the call in the rulebase meaning that is passed around. This lookup operation is performed with respect to the complete call description, *i.e.*, $\mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$. Given the fact that we only record the local component of the contribution of a procedure call (in e, the rulebase meaning), the result of that lookup operation is the local component of the exit description due to the call.

We can repeat the same reasoning as for the natural semantics (Section 5.3.5), showing that for instantiations with Noetherian domains and monotone auxiliary functions, the differential semantics is not only well-defined, but the fixpoint can also be computed by Kleene's sequence.

In the following sections we prove the conditional equivalence of this semantics with the natural semantics we gave earlier.

5.6.1 Conditional Equivalence

In this section we formulate a theorem that relates the goal-dependent semantics to the differential semantics. Indeed, we can show that for some description domains, $\mathbf{P}_M[\![r; q]\!] = \mathbf{P}_{M\delta}[\![r; q]\!]$.

Definition 5.7 *Given the monotonic functions e : Atom → Ans → Ans as computed in $\mathcal{S}em_M$, and e' : Atom → Ans → Ans as computed in $\mathcal{S}em_{M\delta}$, then*

- $e \rightsquigarrow e'$ *iff* $\forall\ (a, \mathcal{S}_0, \mathcal{S}_1) \in e, \exists\ (a, \mathcal{S}_0, \mathcal{S}_l) \in e' : \mathsf{comb}(\mathcal{S}_0, \mathcal{S}_l) = \mathcal{S}_1$.
- $e' \rightsquigarrow e$ *iff* $\forall\ (a, \mathcal{S}_0, \mathcal{S}_l) \in e',\ \exists (a, \mathcal{S}_0, \mathcal{S}_1) \in e : \mathsf{comb}(\mathcal{S}_0, \mathcal{S}_l) = \mathcal{S}_1$.
- $e \sim e'$ *iff* $e \rightsquigarrow e'$ *and* $e' \rightsquigarrow e$.

Definition 5.8 *Let X be a description domain for which the auxiliary functions* init, comb, *and* add *are defined. Let*

$$\begin{array}{rcl} \mathbf{P}^X_M[\![r;q]\!] & = & (\mathcal{S}_M, A_M) \\ \mathbf{P}^X_{M\boldsymbol{\delta}}[\![r;q]\!] & = & (\mathcal{S}_{M\boldsymbol{\delta}}, A_{M\boldsymbol{\delta}}) \end{array}$$

If $\mathcal{S}_M \equiv \mathcal{S}_{M\boldsymbol{\delta}} \wedge A_M \equiv A_{M\boldsymbol{\delta}},\ \forall r \in RuleBase, \forall q$, then the goal-dependent semantics and differential semantics are said to be equivalent *for this domain X. This is denoted by $\mathcal{S}em_M(X) \Leftrightarrow \mathcal{S}em_{M\boldsymbol{\delta}}(X)$.*

Definition 5.9 (Head variable Idempotence) *Let $\mathcal{S} \in Ans$ be a call description for a procedure $p(X_1, \ldots, X_n)$. Assuming that $\mathcal{S}$ respects the strongly moded and typed character of the Mercury language, the* comb *operation is said to be* head variable idempotent *iff*

$$\mathsf{comb}(\mathcal{S},\ \mathcal{S}|_{\{X_1,\ldots,X_n\}}) = \mathcal{S}$$

i.e., *adding the part of a description S that is only related with the head variables of the procedure for which the description describes a call, is a null operation for head variable idempotent* comb *operations.*

Theorem 5.4 *Let X be a complete lattice $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \bot_X, \top_X \rangle$, for which the auxiliary functions $\{\mathsf{init}, \mathsf{comb}, \mathsf{add}\}$[9] are defined. The goal-dependent semantics $\mathcal{S}em_M(X)$ is equivalent to the differential semantics $\mathcal{S}em_{M\boldsymbol{\delta}}(X)$ iff the following statements hold:*

neutral element:
$$\mathsf{comb}(\mathcal{S}, \mathsf{init}) = \mathcal{S} \tag{5.5}$$

associativity:
$$\mathsf{comb}(\mathcal{S}_1, \mathsf{comb}(\mathcal{S}_2, \mathcal{S}_3)) = \mathsf{comb}(\mathsf{comb}(\mathcal{S}_1, \mathcal{S}_2), \mathcal{S}_3), \tag{5.6}$$

additivity:
$$\mathsf{comb}(\mathcal{S}, \mathcal{S}_1 \sqcup_X \mathcal{S}_2) = \mathsf{comb}(\mathcal{S}, \mathcal{S}_1) \sqcup_X \mathsf{comb}(\mathcal{S}, \mathcal{S}_2), \tag{5.7}$$

comb *is head variable idempotent.* (5.8)

add :
$$\mathsf{add}(\mathit{unif}, \mathsf{comb}(\mathcal{S}_1, \mathcal{S}_2)) \quad = \quad \mathsf{comb}(\mathcal{S}_1, \mathsf{add}(\mathit{unif}, \mathcal{S}_2)) \tag{5.9}$$

projection preservation:
$$(\mathsf{comb}(\mathcal{S}_1, \mathcal{S}_2))|_{Vars(\mathcal{S}_1)} = \mathsf{comb}(\mathcal{S}_1,\ \mathcal{S}_2|_{Vars(\mathcal{S}_1)}) \tag{5.10}$$

for all $\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3 \in X$.

Note that comb *does not need to be commutative.*

[9] We omit the explicit superscripts, as they would only make the notation heavier.

Given the fact that the semantics are equivalent only if these conditions are met by the domain instantiating these semantics, we call this *conditional equivalence* and denote it by $\overset{c}{\Leftrightarrow}$. Therefore, here we have: $\mathcal{S}em_M \overset{c}{\Leftrightarrow} \mathcal{S}em_{M\delta}$.

The proof is given by structural induction (Nielson and Nielson 1996; Nielson and Nielson 1992). We split the proof over different levels, namely by first proving the equivalence of the semantics of literals, then goals, and finally procedures. This is done in the following three lemma's. In each of these lemma's we consider a domain $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \bot_X, \top_X \rangle$ with auxiliary functions $\{\mathsf{init}, \mathsf{comb}, \mathsf{add}\}$.

From these equivalences, the global equivalence follows naturally.

In the following lemma's and proofs, we superscribe the different substitution values, rulebase meanings, annotation tables with $\boldsymbol{\gamma}$ to refer to values in the context of the goal-dependent semantics. We use the superscript $\boldsymbol{\delta}$ to refer to these values in the differential semantics.

Lemma 5.1 *Let $\mathcal{S} = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$ and $e^{\delta} \sim e^{\gamma}$. If the conditions (5.5)-(5.10) hold, then:*

$$\begin{array}{rcl} \mathbf{L}_M[\![l]\!]\, e^{\gamma}\mathcal{S} & = & \mathcal{S}^{\gamma} \\ \mathbf{L}_{M\delta}[\![l]\!]\, e^{\delta}\mathcal{S}_g\mathcal{S}_l & = & \mathcal{S}_l^{\delta} \\ & \Downarrow & \\ \mathcal{S}^{\gamma} & = & \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l^{\delta}) \end{array}$$

for all $l \in$ Literal.

Proof We need to prove the equivalence of the exit descriptions. We do this for unifications and procedure calls separately.

- (unification) $\mathcal{S}^{\gamma} = \mathsf{add}(\mathsf{unif}, \mathcal{S})$ and $\mathcal{S}_l^{\delta} = \mathsf{add}(\mathsf{unif}, \mathcal{S}_l)$. Using condition (5.9) we have $\mathcal{S}^{\gamma} = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l^{\delta})$.
- (procedure call) If $\mathcal{S}_{l_p}^{\delta} = e^{\delta}(p(\overline{X}), \mathcal{S})$ and $\mathcal{S}^{\gamma} = e^{\gamma}(p(\overline{X}), \mathcal{S})$ where $\mathcal{S} = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$, then we need to show that

  $$\mathsf{comb}(\mathcal{S}_g, \mathsf{comb}(\mathcal{S}_l, \mathcal{S}_{l_p}^{\delta})) = \mathcal{S}^{\gamma}$$

  Due to the associativity property expressed in Condition 5.6, we have $\mathsf{comb}(\mathcal{S}_g, \mathsf{comb}(\mathcal{S}_l, \mathcal{S}_{l_p}^{\delta})) = \mathsf{comb}(\mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l), \mathcal{S}_{l_p}^{\delta})$ which is equal to $\mathsf{comb}(\mathcal{S}, \mathcal{S}_{l_p}^{\delta})$. As $e^{\gamma} \sim e^{\delta}$, thus $\forall (a, \mathcal{S}, \mathcal{S}') \in e^{\gamma}, \exists (a, \mathcal{S}, \mathcal{S}_l')$ such that $\mathsf{comb}(\mathcal{S}, \mathcal{S}_l') = \mathcal{S}'$, hence in this case $\mathsf{comb}(\mathcal{S}, \mathcal{S}_{l_p}^{\delta}) = \mathcal{S}^{\gamma}$.

Therefore, the lemma holds.

□

Lemma 5.2 *Let $\mathcal{S} = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$, $e^\delta \sim e^\gamma$, and $A^\delta = A^\gamma$. If the conditions (5.5)-(5.10) hold, then:*

$$\begin{array}{rcl} \mathbf{G}_M[\![g]\!](e^\gamma, A^\gamma)\mathcal{S}_g\mathcal{S} & = & (\mathcal{S}^\gamma, A_r^\gamma) \\ \mathbf{G}_{M\delta}[\![g]\!](e^\delta, A^\delta)\mathcal{S}_g\mathcal{S}_l & = & (\mathcal{S}_l^\delta, A_r^\delta) \\ & \Downarrow & \\ \mathcal{S}^\gamma & = & \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l^\delta) \\ A_r^\gamma & = & A_r^\delta \end{array}$$

for all $g \in Goal$.

Proof We prove this lemma for each goal type.

- $g = g_1, g_2$. Applying structural induction twice.
- $g = g_1; g_2$. Using the definitions of $\mathcal{S}em_M$ (Fig. 5.4) and $\mathcal{S}em_{M\delta}$ (Fig. 5.6) we have, by induction:

$$\begin{array}{rcl} \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_{l_1}^\delta) & = & \mathcal{S}_1^\gamma \\ \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_{l_2}^\delta) & = & \mathcal{S}_2^\gamma \\ & \Downarrow & (Cond.\ (5.7)) \\ \mathsf{comb}(\mathcal{S}_g, \sqcup_X\{\mathcal{S}_{l_1}^\delta, \mathcal{S}_{l_2}^\delta\}) & = & \sqcup_X\{\mathcal{S}_1^\gamma, \mathcal{S}_2^\gamma\} \end{array}$$

  Also by induction, we have $A_1^\gamma = A_1^\delta,\ A_2^\gamma = A_2^\delta \Rightarrow A_r^\gamma = A_r^\delta$.
- $g =$ if g_1 then g_2 else g_3. This is similar to the disjunctions. The lemma holds by induction and by the additivity rule (Condition 5.7).
- $g =$ not g_1. Trivial case.
- $g = l$. This is the base case (Lemma 5.1) as far as the correctness of the exit description concerns. The equivalence of the annotation tables follows from the equivalence of the initial annotation tables and that $\mathcal{S} = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l)$.

Therefore, the lemma holds.

□

Lemma 5.3 *Let $e^\delta \sim e^\gamma$, and $A^\delta = A^\gamma$. If the conditions (5.5)-(5.10) hold, then:*

$$\begin{array}{rcl} \mathbf{Pr}_M[\![h \leftarrow g]\!](e^\gamma, A^\gamma)(a, \mathcal{S}) & = & (\mathcal{S}^\gamma, A_r^\gamma) \\ \mathbf{Pr}_{M\delta}[\![h \leftarrow g]\!](e^\delta, A^\delta)(a, \mathcal{S}) & = & (\mathcal{S}_l^\delta, A_r^\delta) \\ & \Downarrow & \\ \mathcal{S}^\gamma & = & \mathsf{comb}(\mathcal{S}, \mathcal{S}_l^\delta) \\ A_r^\gamma & = & A_r^\delta \end{array}$$

for all $(h \leftarrow g) \in Procedure$, where $h = p(X_1, \ldots, X_n)$ and $a = p(Y_1, \ldots, Y_n)$.

Proof We have

$$\begin{array}{l}\mathbf{Pr}_M[\![h \leftarrow g]\!](e^\gamma, A^\gamma)(a, \mathcal{S}) = (\mathcal{S}^\gamma, A_r^\gamma) \\ \quad = \quad \text{let } \mathcal{S}_g = \rho_{a\to h}\left((\mathcal{S})|_a\right) \text{ in} \\ \qquad \text{let } (\mathcal{S}_1^\gamma, A_1^\gamma) = \mathbf{G}_M[\![g]\!](e^\gamma, A^\gamma)\mathcal{S}_g\mathcal{S}_g \text{ in} \\ \qquad\quad (\mathsf{comb}(\mathcal{S}, \rho_{h\to a}\left((\mathcal{S}_1^\gamma)|_h\right)), A_1^\gamma)\end{array}$$

and

$$\begin{array}{l}\mathbf{Pr}_{M\delta}[\![h \leftarrow g]\!](e^\delta, A^\delta)(a, \mathcal{S}) = (\mathcal{S}_l^\delta, A_r^\delta) \\ \quad = \quad \text{let } \mathcal{S}_g = \rho_{a\to h}\left((\mathcal{S})|_a\right) \text{ in} \\ \qquad \text{let } (\mathcal{S}_{l_1}^\delta, A_1^\delta) = \mathbf{G}_{M\delta}[\![g]\!](e^\delta, A^\delta)\mathcal{S}_g\mathsf{init}_{h\leftarrow g} \text{ in} \\ \qquad\quad (\rho_{h\to a}\left(\left(\mathcal{S}_{l_1}^\delta\right)\Big|_h\right), A_1^\delta)\end{array}$$

where we need to prove that $\mathsf{comb}(\mathcal{S}, \mathcal{S}_l^\delta) = \mathcal{S}^\gamma$ and $A_r^\gamma = A_r^\delta$.

By condition (5.5), we have $\mathsf{comb}(\mathcal{S}_g, \mathsf{init}) = \mathcal{S}_g$. We apply Lemma 5.2 and thus obtain that $\mathcal{S}_1^\gamma = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_{l_1}^\delta)$, and $A_1^\gamma = A_1^\delta$ hence $A_r^\gamma = A^{\delta_r}$.

For the equivalence of the descriptions we make the following derivation:

$$\begin{array}{rcl}\mathcal{S}^\gamma & = & \mathsf{comb}(\mathcal{S}, \rho_{h\to a}\left((\mathcal{S}_1^\gamma)|_h\right)) \\ & \Downarrow & \quad \text{Induction: } \mathcal{S}_1^\gamma = \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_{l_1}^\delta) \\ & = & \mathsf{comb}(\mathcal{S}, \rho_{h\to a}\left(\left(\mathsf{comb}(\mathcal{S}_g, \mathcal{S}_{l_1}^\delta)\right)\Big|_h\right)) \\ & \Downarrow & \quad \mathcal{S}_g = \rho_{a\to h}\left((\mathcal{S})|_a\right) \\ & = & \mathsf{comb}(\mathcal{S}, \rho_{h\to a}\left(\left(\mathsf{comb}(\rho_{a\to h}\left(\mathcal{S}|_a\right), \mathcal{S}_{l_1}^\delta)\right)\Big|_h\right)) \\ & \Downarrow & \quad \rho_{h\to a}\left((\mathcal{S})|_h\right) = \left(\rho_{h\to a}\left(\mathcal{S}\right)\right)|_a, \quad \forall \mathcal{S} \\ & = & \mathsf{comb}(\mathcal{S}, \left(\rho_{h\to a}\left(\mathsf{comb}(\rho_{a\to h}\left((\mathcal{S})|_a\right), \mathcal{S}_{l_1}^\delta)\right)\right)\Big|_a) \\ & \Downarrow & \quad \rho_{h\to a}\left(\mathsf{comb}(\mathcal{S}_1, \mathcal{S}_2)\right) = \mathsf{comb}(\rho_{h\to a}\left(\mathcal{S}_1\right), \rho_{h\to a}\left(\mathcal{S}_2\right)), \quad \forall \mathcal{S}_1, \mathcal{S}_2 \\ & = & \mathsf{comb}(\mathcal{S}, \left(\mathsf{comb}(\rho_{h\to a}\left(\rho_{a\to h}\left(\mathcal{S}|_a\right)\right), \rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\right)\right)\right)\Big|_a) \\ & \Downarrow & \quad \rho_{h\to a}\left(\rho_{a\to h}\left(\mathcal{S}\right)\right) = \mathcal{S}, \quad \forall \mathcal{S} \\ & = & \mathsf{comb}(\mathcal{S}, \left(\mathsf{comb}(\mathcal{S}|_a, \rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\right)\right)\right)\Big|_a) \\ & \Downarrow & \quad \text{Condition 5.10} \\ & = & \mathsf{comb}(\mathcal{S}, \mathsf{comb}(\mathcal{S}|_a, \left(\rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\right)\right)\Big|_a)) \\ & \Downarrow & \quad \text{Condition 5.6} \\ & = & \mathsf{comb}(\mathsf{comb}(\mathcal{S}, \mathcal{S}|_a), \left(\rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\right)\right)\Big|_a) \\ & \Downarrow & \quad \text{Condition 5.8} \\ & = & \mathsf{comb}(\mathcal{S}, \left(\rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\right)\right)\Big|_a) \\ & \Downarrow & \quad \left(\rho_{h\to a}\left(\mathcal{S}\right)\right)|_a = \rho_{h\to a}\left(\mathcal{S}|_h\right), \quad \forall \mathcal{S} \\ & = & \mathsf{comb}(\mathcal{S}, \rho_{h\to a}\left(\mathcal{S}_{l_1}^\delta\Big|_h\right)) \\ & = & \mathsf{comb}(\mathcal{S}, \mathcal{S}_l^\delta)\end{array}$$

Therefore $\mathcal{S}^{\gamma} = \mathsf{comb}(\mathcal{S}, \mathcal{S}_l^{\delta})$, which proves the lemma.

□

We can now prove Theorem 5.4.

Proof (Theorem 5.4) By the fact that for each goal and for each literal we can relate its goal-dependent semantics to its differential semantics by the simple relation $\mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l) = \mathcal{S}$, we have exactly the same procedure calls in both semantics, and therefore, at each iteration $e^{\gamma} \sim e^{\delta}$ and $A^{\gamma} = A^{\delta}$. If a fixpoint is ever reached — which depends on the auxiliary operations being monotone or continuous in the given description domain — then $\mathbf{R}_M[\![r]\!] \sim \mathbf{R}_{M\delta}[\![r]\!]$. Given the fact that $\mathsf{comb}(\mathsf{init}, \mathsf{init}) = \mathsf{init}$, we can apply Lemma 5.2. Therefore $\mathbf{P}_M[\![r;\, q]\!] = \mathbf{P}_{M\delta}[\![r;\, q]\!]$, $\forall r \in \mathit{Program}$.

□

The two last conditions, Condition 5.9 on the auxiliary function add and Condition 5.10 on the comb operation related to the projection operation, may seem odd at first sight. In the next section we show that these conditions are nevertheless easily met by the concrete description domain we used until now. In Chapter 6 we show that even the extended concrete domain that we use in the context of structure sharing satisfies these conditions.

5.6.2 Concrete Differential Semantics

It is interesting to show that the differential semantics instantiated with the concrete domain $\wp(\mathit{Eqn}^{+})$ is equivalent to the goal-dependent Mercury semantics instantiated with that domain. We prove this in this section.

Definition 5.10 (Concrete Differential Semantics) *Let Ans be the concrete domain of existentially quantified ex-equations $\wp(\mathit{Eqn}^{+})$ as used in the concrete goal-dependent semantics for Simple Mercury (page 66) and for Mercury (page 78), where the combination operation* comb *corresponds to the combination operation defined earlier in the context of Simple Mercury (Definition 5.6), then the differential semantics instantiated with this domain is called the* concrete differential semantics *of Mercury, denoted by $\mathcal{S}em_{M\delta}(\wp(\mathit{Eqn}^{+}))$.*

Theorem 5.5 *The concrete differential semantics $\mathcal{S}em_{M\delta}(\wp(\mathit{Eqn}^{+}))$ is equivalent to the concrete goal-dependent semantics $\mathcal{S}em_{M}(\wp(\mathit{Eqn}^{+}))$.*

Proof We show that $\mathsf{init}^{\wp(\mathit{Eqn}^{+})}$, $\mathsf{comb}^{\wp(\mathit{Eqn}^{+})}$ and $\mathsf{add}^{\wp(\mathit{Eqn}^{+})}$ satisfy the required properties. In the following we abbreviate these functions to init, comb and add resp.

1. (Condition 5.5) init *is neutral w.r.t.* comb: $\forall E \in \wp(Eqn^+)$ we have $\mathsf{comb}(E, \mathsf{init}) = \mathsf{comb}(E, \{\mathsf{true}\}) = \{e \wedge \mathsf{true} \mid e \in E\} = \{e \mid e \in E\} = E$.

2. (Condition 5.6) comb *is associative*: this comes as a direct consequence of the associativity of the boolean $\wedge$ operation.

3. (Condition 5.7) comb *is additive*: $\forall E, E_1, E_2 \in \wp(Eqn^+)$ we can write the derivation $\mathsf{comb}(E, E_1 \cup E_2) = \{e \wedge e' \mid e \in E \wedge (e' \in E_1 \vee e' \in E_2\} = \{e \wedge e_1 \mid e \in E \wedge e_1 \in E_1\} \cup \{e \wedge e_2 \mid e \in E \wedge e_2 \in E_2\} = \mathsf{comb}(E, E_1) \cup \mathsf{comb}(E, E_2)$.

4. (Condition 5.8) comb *is head variable idempotent*. Consider a procedure call $p(X_1, \ldots, X_n)$ where $\mathcal{H} = \{X_1, \ldots, X_n\}$ represents the head variables of the procedure call. Let $E \in \wp(Eqn^+)$ be a valid call description for $p(X_1, \ldots, X_n)$. Let $E = \{e_1, \ldots, e_k\}$ and $E|_{\mathcal{H}} = \{e'_1, \ldots, e'_l\}$.

   E is a call description, therefore it must conform to the mode-information associated with the called procedure. This means that after projection on the head variables, all variables in the resulting description must have the same degree of instantiatedness. Knowing that Mercury does not allow partially instantiated data structures, this means that the solved form for each constraint set e in $E|_{\mathcal{H}}$ will always involve the same set of variables, *i.e.*, $\forall e_1, e_2 \in E|_{\mathcal{H}} : \mathit{Vars}(e_1) = \mathit{Vars}(e_2)$. This means that in order to have different constraint sets e_1, e_2 in $E|_{\mathcal{H}}$, at least one variable must have a different value attributed: $\forall e_1, e_2 \in E|_{\mathcal{H}} \wedge e_1 \neq e_2 : \exists X \in \mathit{Vars}(e_1) : e_1 \models X = t_i \wedge e_2 \models X = t_j \wedge t_i \neq t_j$ Given this situation, this implies that $\forall e_1, e_2 \in E|_{\mathcal{H}} \wedge e_1 \neq e_2 : e_1 \wedge e_2 \models \mathsf{false}$.

   Relating the constraints from E to $E|_{\mathcal{H}}$ we have one of the following situations. Let $e \in E$, and $e' \in E|_{\mathcal{H}}$, then

   - either $e \models e'$, in which case $e \wedge e' = e$
   - or $e \not\models e'$, which means that there exists a variable $X \in \mathit{Vars}(e')$ such that $e \models X = t_1, e' \models X = t_2$ but where $t_1 \neq t_2$, and therefore $e \wedge e' \models \mathsf{false}$.

   Given these two possibilities, we can conclude that

   $$\mathsf{comb}(E, E|_{\mathcal{H}}) = \{e \wedge e' \mid e \in E, e' \in E|_{\mathcal{H}}\} = E$$

   hence, comb is head variable idempotent.

5. (Condition 5.9) *Adding a unification to a combination of descriptions is equivalent to combining the descriptions with the added unification:* $\forall E_1, E_2 \in$

$\wp(Eqn^+)$ and $\forall \text{unif} \in Prim$ we have

$$\begin{array}{lll} & \mathsf{add}(\text{unif}, \mathsf{comb}(E_1, E_2)) & \\ = & \mathsf{comb}(\{\text{unif}\}, \mathsf{comb}(E_1, E_2)) & \\ = & \mathsf{comb}(\{\text{unif}\}, \mathsf{comb}(E_2, E_1)) & \text{(commutativity)} \\ = & \mathsf{comb}(\mathsf{comb}(\{\text{unif}\}, E_2), E_1) & \text{(associativity)} \\ = & \mathsf{comb}(\mathsf{add}(\text{unif}, E_2), E_1) & \\ = & \mathsf{comb}(E_1, \mathsf{add}(\text{unif}, E_2)) & \text{(commutativity)} \end{array}$$

6. (Condition 5.10) $\forall E_1, E_2 \in \wp(Eqn^+)$, let $V = Vars(E_1)$, then

$$\begin{array}{rcl} (\mathsf{comb}(E_1, E_2))|_V & = & (\{e_1 \wedge e_2 \,|\, e_1 \in E_1 \wedge e_2 \in E_2\})|_V \\ & = & \{\overline{\exists}_V(e_1 \wedge e_2) \,|\, e_1 \in E_1 \wedge e_2 \in E_2\} \end{array}$$

Given the fact that $\forall e_1 \in E_1$ we have $Vars(e_1) \subseteq V$ therefore $\overline{\exists}_V(e_1 \wedge e) \Leftrightarrow e_1 \wedge \overline{\exists}_V e$, $\forall e \in \wp(Eqn^+)$. Hence, $\{\overline{\exists}_V(e_1 \wedge e_2) \,|\, e_1 \in E_1 \wedge e_2 \in E_2\} = \{e_1 \wedge \overline{\exists}_V e_2 \,|\, e_1 \in E_1 \wedge e_2 \in E_2\} = \{e_1 \wedge e_2' \,|\, e_1 \in E_1 \wedge e_2' \in (E_2)|_V\} = \mathsf{comb}(E_1, (E_2)|_V)$

All conditions from Theorem 5.4 are satisfied, hence we can conclude that the semantics $\mathcal{S}em_{M\delta}(\wp(Eqn^+))$ is equivalent to the concrete goal-dependent semantics $\mathcal{S}em_M(\wp(Eqn^+))$.

□

5.6.3 Abstract Differential Semantics, Relative Precision

In the previous section we demonstrated the equivalence between the differential concrete semantics $\mathcal{S}em_{M\delta}(\wp(Eqn^+))$ and the instantiated natural semantics $\mathcal{S}em_M(\wp(Eqn^+))$. In the following chapters, we extend our concrete domain to comprise other characteristics of the variables than only their bindings, yet we again show that for these extended domains, the natural interpretation of the language remains equivalent to the differential semantics. This is an essential step to be able to relate our abstract semantics (which we plan to define only in the context of the differential semantics, as it allows the straightforward step to the goal-independent based semantics) to the original natural semantics of the language, as the abstract domains we will be using do not satisfy the equivalence conditions required by Theorem 5.4.

Note that if instantiations of the natural and differential semantics do not meet the equivalence conditions stated by Theorem 5.4, then in general, it is impossible to relate the results obtained for each of these instantiations: for some instantiations this may mean that the descriptions obtained in the differential semantics will always approximate the descriptions in the natural semantics; for other instantiations, the approximating relation may work the other way around; and

even for other instantiations, there may be no general approximating relation at all.

The most demanding equivalence condition of Theorem 5.4 seems to be the required idempotence of the combination operator. If that is indeed the only violation of that theorem, then it can be proved that when instantiated with that domain, the natural semantics will be less precise than the concrete semantics. This will be the case for the abstract domains we intend to use, more precisely, the domain of abstract structure sharing (c.f. Chapter 6), where the combination operation relies on the alternating closure of two sets, an operation known not to be idempotent. This is therefore yet another argument for defining our abstract semantics in the differential context.

Although this is an interesting aspect of the semantics developed in this thesis, we will not detail it further.

5.6.4 Implementation Issues

As the presented differential semantics is still goal directed, the same reasoning as for the goal-dependent semantics $\mathcal{S}em_S$ and $\mathcal{S}em_M$ holds (page 72). Hence, the most natural way of implementing this semantics is a top-down lazy evaluation scheme.

5.7 Goal-Independent Based Semantics

Here we present a goal-independent based semantics for Mercury programs. This semantics consists of two parts: a set of clauses giving a goal-independent meaning to the individual procedures in the rulebase of a program, and a set of clauses that gives a goal-dependent meaning to a query q in the context of that program using the goal-independent meaning defined by the former clauses.

Figure 5.9 presents the functions defining the goal-independent meaning of a rulebase. The types and signatures of these functions are shown in Figure 5.8. The goal-dependent meaning of a program, based on the goal-independent meaning of its rulebase, is presented in Figure 5.11. Figure 5.10 presents the signatures of the functions used in Figure 5.11.

The goal-independent meaning of a rulebase is seen as a table that maps each procedure (identified by a corresponding call atom) to an exit description. Goal-independence is here achieved by assuming that each procedure is called with an empty call description, *i.e.*, assuming that all variables are unbound. In the context of a logic programming language, this represents a viable initial situation allowing the derivation of goal-independent information. Thus, the exit description is obtained by computing the meaning of the procedure assuming that it is called with the empty call description (returned by the auxiliary function $\mathsf{init}_{h\leftarrow g}$). The semantics of Mercury goals is therefore defined in a goal-dependent way,

$$
\begin{array}{lcl}
\mathit{GIProcMeaning} & = & \mathit{Procedure} \to \mathit{Ans} \\
\mathbf{R}_{M\star} & : & \mathit{RuleBase} \to \mathit{GIProcMeaning} \\
\mathbf{F}_{M\star} & : & \mathit{RuleBase} \to \mathit{GIProcMeaning} \to \mathit{GIProcMeaning} \\
 & \equiv & \mathit{RuleBase} \to \mathit{GIProcMeaning} \to \mathit{Atom} \to \mathit{Ans} \\
\mathbf{Pr}_{M\star} & : & \mathit{Procedure} \to \mathit{GIProcMeaning} \to \mathit{Atom} \to \mathit{Ans} \\
\mathbf{G}_{M\star} & : & \mathit{Goal} \to \mathit{GIProcMeaning} \to \mathit{Ans} \to \mathit{Ans} \\
\mathbf{L}_{M\star} & : & \mathit{Literal} \to \mathit{GIProcMeaning} \to \mathit{Ans} \to \mathit{Ans}
\end{array}
$$

Figure 5.8: Types and Signatures of the semantic functions used in $\mathcal{S}em_{M\star}$.

$$
\begin{array}{lcl}
\mathbf{R}_{M\star}[\![r]\!] & = & \mathrm{fix}(\mathbf{F}_{M\star}[\![r]\!]) \\
\mathbf{F}_{M\star}[\![p_1 \ldots p_i \ldots p_{n_p}]\!] e p_i(\overline{Y}) & = & \mathbf{Pr}_{M\star}[\![p_i]\!] e p_i(\overline{Y}) \\
\mathbf{Pr}_{M\star}[\![h \leftarrow g]\!] e a & = & \rho_{h\to a}\left(\left(\mathbf{G}_{M\star}[\![g]\!] e\, \mathsf{init}_{h\leftarrow g}\right)\Big|_h\right) \\
\mathbf{G}_{M\star}[\![g_1, g_2]\!] e\mathcal{S} & = & \mathbf{G}_{M\star}[\![g_2]\!] e(\mathbf{G}_{M\star}[\![g_1]\!] e\mathcal{S}) \\
\mathbf{G}_{M\star}[\![g_1; g_2]\!] e\mathcal{S} & = & \text{let } \mathcal{S}_1 = \mathbf{G}_{M\star}[\![g_1]\!] e\mathcal{S} \text{ in} \\
 & & \text{let } \mathcal{S}_2 = \mathbf{G}_{M\star}[\![g_2]\!] e\mathcal{S} \text{ in} \\
 & & \quad \mathcal{S}_1 \sqcup \mathcal{S}_2 \\
\mathbf{G}_{M\star}[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!] e\mathcal{S} & = & \text{let } \mathcal{S}_1 = \mathbf{G}_{M\star}[\![g_1]\!] e\mathcal{S} \text{ in} \\
 & & \text{let } \mathcal{S}_2 = \mathbf{G}_{M\star}[\![g_2]\!] e\mathcal{S}_1 \text{ in} \\
 & & \text{let } \mathcal{S}_3 = \mathbf{G}_{M\star}[\![g_2]\!] e\mathcal{S} \text{ in} \\
 & & \quad \mathcal{S}_2 \sqcup \mathcal{S}_3 \\
\mathbf{G}_{M\star}[\![\text{not } g]\!] e\mathcal{S} & = & \mathcal{S} \\
\mathbf{G}_{M\star}[\![l]\!] e\mathcal{S} & = & \mathbf{L}_{M\star}[\![l]\!]\, e\, \mathcal{S} \\
\mathbf{L}_{M\star}[\![\text{unif}]\!]\, e\, \mathcal{S} & = & \mathsf{add}(\text{unif}, \mathcal{S}) \\
\mathbf{L}_{M\star}[\![p(\overline{X})]\!]\, e\, \mathcal{S} & = & \mathsf{comb}(\mathcal{S}, e(p(\overline{X})))
\end{array}
$$

Figure 5.9: Clauses defining the goal-independent semantics $\mathcal{S}em_{M\star}$ of a Mercury rulebase.

thus similar to its definition in the context of the goal-dependent semantics for Mercury. The only difference is that no annotation table is built. The meaning for unifications is, as usual, the result of adding the unification to the call description of that literal. Finally, the meaning of a procedure call is defined as the combination of the current call description with the stored goal-independent exit description.

In this way, each procedure is given a meaning with the intention that this meaning represents the contribution of that procedure to any call description with which it might be called. Instead of computing this local contribution with respect to a specific call pattern of a procedure, in this approach we define this contribution independently.

The goal-independent based semantics of a Mercury program can now be

defined in terms of the goal-independent meaning of its rulebase. This is what is represented in Figure 5.11, where $e^\star$ represents the goal-independent meaning of the rulebase.

$$
\begin{array}{lcl}
\mathbf{P}_{M\bullet} & : & \mathit{Program} \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{R}_{M\bullet} & : & \mathit{RuleBase} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{AProcMeaning} \\
\mathbf{F}_{M\bullet} & : & \mathit{RuleBase} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{AProcMeaning} \\
\mathbf{Pr}_{M\bullet} & : & \mathit{Procedure} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{AProcMeaning} \\
\mathbf{G}_{M\bullet} & : & \mathit{Goal} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{AProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans} \\
 & & \quad \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{L}_{M\bullet} & : & \mathit{Literal} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans}
\end{array}
$$

Figure 5.10: Signatures of the semantic functions used in $\mathcal{S}em_{M\bullet}$.

In this semantics, the meaning of a program is defined as the goal-dependent meaning of the query q, with respect to the goal-dependent annotated rulebase meaning which itself is defined in terms of the goal-independent meaning of the procedures within the rulebase ($\mathbf{R}_{M\star}[\![r]\!]$). Note, that just as for the differential semantics, only the local contributions of the exit descriptions are computed and recorded.

The clauses defining the semantics of a goal or literal are the same as the ones used in the context of the differential semantics. The reason is that in this semantics, we also construct the local contributions of the exit descriptions, based on the already precomputed goal-dependent rule-base meaning e. The main difference of this semantics w.r.t. $\mathcal{S}em_{M\delta}$ is the definition of the semantics of procedures, *i.e.*, $\mathbf{Pr}_{M\bullet}$. This is the only clause that uses the goal-independent meaning of the rulebase. Instead of computing the local component of the exit description of the procedure based on the results of interpreting the goal of the procedure, this clause uses the precomputed goal-independent meaning of the procedure. As this result will always be the same, for each consecutive call of the procedure, we obtain the effect that the exit descriptions are not part of the fixpoint process in this semantics. The only values really involved in the fixpoint function are the goal-dependent annotations. Given the fact that these annotations are based on the already precomputed goal-independent meanings (as the exit descriptions of each procedure is based on these meanings) we can conclude that one iteration over the called procedures is sufficient. Hence, the fixpoint function stabilises after one single iteration.

In the following section we discuss the equivalence of the goal-independent based semantics $\mathcal{S}em_{M\bullet}$ with the differential semantics $\mathcal{S}em_{M\delta}$. We also discuss some implementation issues.

$$
\begin{array}{lcl}
\mathbf{P}_{M\bullet}[\![r;q]\!] & = & \text{let } \iota = \mathsf{init}_q \text{ in} \\
 & & \quad \mathbf{G}_{M\bullet}[\![q]\!](\mathbf{R}_{M\bullet}[\![r]\!](\mathbf{R}_{M\star}[\![r]\!]))\iota\,\iota \\
\mathbf{R}_{M\bullet}[\![r]\!]\,e^\star & = & \mathrm{fix}(\mathbf{F}_{M\bullet}[\![r]\!]\,e^\star) \\
\mathbf{F}_{M\bullet}[\![p_1 \ldots p_i \ldots p_{n_p}]\!]e^\star(e,A)p_i(\overline{Y})\mathcal{S} & & \\
 & = & \mathbf{Pr}_{M\bullet}[\![p_i]\!]e^\star(e,A)p_i(\overline{Y})\mathcal{S} \\
\mathbf{Pr}_{M\bullet}[\![h \leftarrow g]\!]e^\star(e,A)a\mathcal{S} & = & \text{let } \mathcal{S}_g = \rho_{a\to h}\,((\mathcal{S})|_a) \text{ in} \\
 & & \quad \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_{M\bullet}[\![g]\!](e,A)\mathcal{S}_g\mathsf{init}_{h\leftarrow g} \text{ in} \\
 & & \quad\quad (e^\star a, A_1) \\
\mathbf{G}_{M\bullet}[\![g_1,g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\bullet}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad \mathbf{G}_{M\bullet}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1} \\
\mathbf{G}_{M\bullet}[\![g_1;g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\bullet}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M\bullet}[\![g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_{l_1} \sqcup \mathcal{S}_{l_2}, \mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_{M\bullet}[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l & & \\
 & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\bullet}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M\bullet}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1} \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_3}, A_3) = \mathbf{G}_{M\bullet}[\![g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_{l_2} \sqcup \mathcal{S}_{l_3}, \mathsf{merge}(A_2,A_3)) \\
\mathbf{G}_{M\bullet}[\![\text{not } g]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M\bullet}[\![g]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_l, A_1) \\
\mathbf{G}_{M\bullet}[\![l]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & (\mathbf{L}_{M\bullet}[\![l]\!]\,e\,\mathcal{S}_g\mathcal{S}_l, A[(\mathsf{pp}(l),\mathcal{S}_g), \mathsf{comb}(\mathcal{S}_g,\mathcal{S}_l)]) \\
\mathbf{L}_{M\bullet}[\![\text{unif}]\!]\,e\,\mathcal{S}_g\mathcal{S}_l & = & \mathsf{add}(\text{unif}, \mathcal{S}_l) \\
\mathbf{L}_{M\bullet}[\![p(\overline{X})]\!]\,e\,\mathcal{S}_g\mathcal{S}_l & = & \mathsf{comb}(\mathcal{S}_l, e(p(\overline{X}), \mathsf{comb}(\mathcal{S}_g,\mathcal{S}_l)))
\end{array}
$$

Figure 5.11: Clauses defining the goal-independent based meaning $\mathcal{S}em_{M\bullet}$ of a Mercury program. We use the notation $e^\star$ to denote the goal-independent rulebase meaning of the rulebase of the program.

5.7.1 Equivalence

Independent from the description domain, the interpretation of a Mercury program under the differential semantics is always equivalent to its interpretation in the goal-independent based semantics. Intuitively we can see that the definition of the local exit descriptions in the definition of $\mathcal{S}em_{M\delta}$ is independent of the exact call description of the procedure, hence it should be equal to the goal-independent meaning that we have defined for a rulebase. Moreover, the goal-independent based semantics $\mathcal{S}em_{M\bullet}$ only records the local components of the exit descriptions in its goal-dependent rulebase meaning e. This corresponds to what is recorded in the goal-dependent rulebase meaning in the context of the differential semantics. We may therefore conclude that these meanings will also

be equivalent. Finally, the use of the call descriptions in the definition of the individual annotations is the same in the definition of $\mathcal{S}em_{M\bullet}$ as in $\mathcal{S}em_{M\delta}$, so we can expect that also the annotations are equivalent.

In the following paragraphs we prove this equivalence formally.

For this purpose, we show that the local components of the exit descriptions defined in $\mathcal{S}em_{M\delta}$ are independent of the exact call descriptions. This call description independence is stated by the following theorem.

Lemma 5.4 *Let*

$$\begin{array}{rcl}(\mathcal{S}_{l_1}, A_1) & = & \mathbf{Pr}_{M\delta}[\![h \leftarrow g]\!](e, A)(a_1, \mathcal{S}_1) \\ (\mathcal{S}_{l_2}, A_2) & = & \mathbf{Pr}_{M\delta}[\![h \leftarrow g]\!](e, A)(a_2, \mathcal{S}_2)\end{array}$$

where e is a rulebase meaning such that

$$\forall (p_1, \mathcal{S}_{g_1}, \mathcal{S}_{l_1}), (p_2, \mathcal{S}_{g_2}, \mathcal{S}_{l_2}) \in e : p_1 = \rho\,(p_2) \Rightarrow \mathcal{S}_{l_1} = \rho\,(\mathcal{S}_{l_2}) \tag{5.11}$$

If $a_1 = \rho\,(a_2)$, then $\mathcal{S}_{l_1} = \rho\,(\mathcal{S}_{l_2})$ where ρ is the renaming function mapping variables of a_1 to the corresponding variables of a_2.

Proof Just as we did in the (conditional) equivalence proof of $\mathcal{S}em_M$ and $\mathcal{S}em_{M\delta}$, we develop intermediate proofs for each individual clause in the definition of $\mathcal{S}em_{M\delta}$.

1. **(Base case)** The local components of the exit descriptions obtained with $\mathbf{L}_{M\delta}$ are independent of the call description of the procedure to which it belongs to. Let

$$\begin{array}{rcl}\mathcal{S}_{l_1} & = & \mathbf{L}_{M\delta}[\![l]\!]\, e\, \mathcal{S}_{g_1} \mathcal{S}_l \\ \mathcal{S}_{l_2} & = & \mathbf{L}_{M\delta}[\![l]\!]\, e\, \mathcal{S}_{g_2} \mathcal{S}_l\end{array}$$

then $\forall \mathcal{S}_{g_1}, \mathcal{S}_{g_2}$ we must have $\mathcal{S}_{l_1} = \mathcal{S}_{l_2}$.
Proof:
ⓐ if $l = \mathsf{unif}$, then

$$\begin{array}{rcl}\mathcal{S}_{l_1} & = & \mathsf{add}(\mathsf{unif}, \mathcal{S}_l) \\ \mathcal{S}_{l_2} & = & \mathsf{add}(\mathsf{unif}, \mathcal{S}_l)\end{array}$$

Obviously, $\mathcal{S}_{l_1} = \mathcal{S}_{l_2}$.
ⓑ if $l = p(\overline{X})$, then

$$\begin{array}{rcl}\mathcal{S}_{l_1} & = & \mathsf{comb}(\mathcal{S}_l, e(p(\overline{X}), \mathcal{S}_{c_1})) \\ \mathcal{S}_{l_2} & = & \mathsf{comb}(\mathcal{S}_l, e(p(\overline{X}), \mathcal{S}_{c_2}))\end{array}$$

where

$$\begin{array}{rcl}\mathcal{S}_{c_1} & = & \mathsf{comb}(\mathcal{S}_{g_1}, \mathcal{S}_l) \\ \mathcal{S}_{c_2} & = & \mathsf{comb}(\mathcal{S}_{g_2}, \mathcal{S}_l)\end{array}$$

As we assume Equation 5.11, we have that $e(p(\overline{X}), \mathcal{S}_{c_1}) = e(p(\overline{X}), \mathcal{S}_{c_2}) = \mathcal{S}_{l_p}, \forall \mathcal{S}_{c_1}, \mathcal{S}_{c_2}$. Hence

$$\begin{array}{rcl} \mathcal{S}_{l_1} & = & \mathsf{comb}(\mathcal{S}_l, \mathcal{S}_{l_p}) \\ \mathcal{S}_{l_2} & = & \mathsf{comb}(\mathcal{S}_l, \mathcal{S}_{l_p}) \end{array}$$

and therefore $\mathcal{S}_{l_1} = \mathcal{S}_{l_2}, \forall \mathcal{S}_{g_1}, \mathcal{S}_{g_2}$.

2. The local components of the exit descriptions obtained with $\mathbf{G}_{M\delta}$ are independent of the call description of the procedure to which it belongs to. Let

$$\begin{array}{rcl} (\mathcal{S}_{l_1}, A_1) & = & \mathbf{G}_{M\delta}[\![g]\!](e, A)\mathcal{S}_{g_1}\mathcal{S}_l \\ (\mathcal{S}_{l_2}, A_2) & = & \mathbf{G}_{M\delta}[\![g]\!](e, A)\mathcal{S}_{g_2}\mathcal{S}_l \end{array}$$

then $\forall \mathcal{S}_{g_1}, \mathcal{S}_{g_2}$ we have $\mathcal{S}_{l_1} = \mathcal{S}_{l_2}$.
Proof: For goals composed out of a single literal, we fall back on the base case. For all the other types of goals, the above statement can be proved by induction.

Finally, based on the previous statements, it is trivial to show that using the definition of $\mathbf{Pr}_{M\delta}$, the obtained local component of the exit description is also independent from the exact call description $\mathcal{S}$.

□

Lemma 5.5 *Let e be the differential rulebase meaning of a rulebase r, thus* $e = \mathbf{R}_{M\delta}[\![r]\!]$, *then e satisfies Equation 5.11 of Lemma 5.4.*

Proof The rulebase meaning is defined as the least fixpoint over $\mathbf{F}_{M\delta}$, which is defined in terms of $\mathbf{Pr}_{M\delta}$. As we assume Noetherian description domains with monotone auxiliary operations, we can compute this fixpoint as a Kleene sequence. Let $e^0, e^1, \ldots, e^(i-1), e^i, \ldots$ be that sequence. We can show that for each consecutive rulebase meaning in that sequence, Equation 5.11 holds. Indeed, it holds for the initial rulebase meaning $e^0 = \bot$. Assuming that it holds for $e^(i-1)$ we show that it holds for e^i too, and therefore, by induction, it holds for every rulebase meaning in that sequence. As Equation 5.11 holds for $e^(i-1)$ we can apply Lemma 5.4. This means means that whenever the semantics of a procedure is computed for different call descriptions, the resulting local contributions will always be identical, modulo possible renaming. This means that the new rulebase meaning computed based on the old rulebase meaning will also satisfy Equation 5.11.

□

Given the fact that the clauses of the semantics of procedures, goals and literals in $\mathcal{S}em_{M\delta}$ are similar to the clauses of procedures, goals and literals in $\mathcal{S}em_{M\star}$ w.r.t. the derived exit descriptions, it is correct to conclude that these exit descriptions are the same. Or put differently, the goal-independent semantics of a rulebase is equivalent to the local component of the exit descriptions of the rulebase as defined by the differential semantics for our language. This is expressed in the following lemma:

Lemma 5.6 *Let* $e^\star = \mathbf{R}_{M\star}[\![r]\!]$*, and* $e^\delta = \mathbf{R}_{M\delta}[\![r]\!]$*, then for every entry in* $e^\star$*,* i.e.*,* $\forall(p_1, \mathcal{S}^\star) \in e^\star$ *and every entry in* e^δ*,* i.e.*,* $\forall(p_2, \mathcal{S}_g^\delta, \mathcal{S}_l^\delta) \in e^\delta$ *we have* $p_1 = \rho(p_2) \Rightarrow \mathcal{S}^\star = \rho(\mathcal{S}_l^\delta)$ *where* ρ *is the renaming function mapping variables of* p_2 *on the corresponding variables of* p_1*.*

Given the fact that the exit description of a procedure in $\mathcal{S}em_{M\bullet}$ is equal to the goal-independent exit description of that procedure, the following lemma holds:

Lemma 5.7 *Let* $e^\star = \mathbf{R}_{M\star}[\![r]\!]$*, and* $e^\bullet = \mathbf{R}_{M\bullet}[\![r]\!]e^\star$*, then for every entry in* $e^\star$*,* i.e.*,* $\forall(p_1, \mathcal{S}^\star) \in e^\star$ *and for every entry in* $e^\bullet$*,* i.e.*,* $\forall(p_2, \mathcal{S}_g^\bullet, \mathcal{S}_l^\bullet) \in e^\bullet$ *we have* $p_1 = \rho(p_2) \Rightarrow \mathcal{S}^\star = \rho(\mathcal{S}_l^\bullet)$ *where* ρ *is the renaming function mapping variables of* p_2 *on the corresponding variables of* p_1*.*

By transitivity, we therefore have:

Lemma 5.8 *Let*

$$\begin{array}{rcl} e^\star & = & \mathbf{R}_{M\star}[\![r]\!] \\ e^\bullet & = & \mathbf{R}_{M\bullet}[\![r]\!]e^\star \\ e^\delta & = & \mathbf{R}_{M\delta}[\![r]\!] \end{array}$$

then $\forall(p_1, \mathcal{S}_g^\bullet, \mathcal{S}_l^\bullet) \in e^\bullet$ *and* $\forall(p_2, \mathcal{S}_g^\delta, \mathcal{S}_l^\delta) \in e^\delta : p_1 = \rho(p_2) \Rightarrow \mathcal{S}_l^\bullet = \rho(\mathcal{S}_l^\delta)$ *where* ρ *is the renaming function mapping variables of* p_2 *on the corresponding variables of* p_1*.*

As a consequence of the previous lemmas, we can correctly conclude that the result of interpreting a program in $\mathcal{S}em_{M\delta}$ is equivalent to the result of interpreting that same program in $\mathcal{S}em_{M\bullet}$. This means that the exit descriptions of the query are equal in both semantics as well as the generated annotations.

Theorem 5.6 *Let*

$$\begin{array}{rcl} (\mathcal{S}^\delta, A^\delta) & = & \mathbf{P}_{M\delta}[\![r; q]\!] \\ (\mathcal{S}^\bullet, A^\bullet) & = & \mathbf{P}_{M\bullet}[\![r; q]\!] \end{array}$$

then $\mathcal{S}^\delta = \mathcal{S}^\bullet$ *and* $A^\delta = A^\bullet$*.*

Proof Indeed, the annotations are computed based on the rulebase meaning e that is computed based on the goal-independent rulebase meaning of the program. This rulebase meaning is equivalent to the rulebase meaning

constructed in $\mathcal{S}em_{M\delta}$, therefore, by the similarity of the different clauses defining $\mathcal{S}em_{M\bullet}$, we can conclude that both semantics are equivalent.

□

Note that this equivalence is independent of the description domain. No additional constraints are imposed on the domain.

5.7.2 Implementation Issues

The goal-independent based semantics is defined in two parts: a rulebase is given a goal-independent meaning, upon which the program is given its goal-dependent meaning. Intuitively, this definition of the semantics suggests a bottom/up implementation where the procedures are given a goal-independent meaning in isolation followed by a top/down execution scheme in the presence of an actual query. However, in the concrete domain, this may not be the recommended way of implementing the language, as the goal-independent exit description for procedures may in general be infinite.

5.8 Adding Pre-Annotations

In the differential semantics we separated the exit descriptions of procedures into their global component, and their local component. In the definition of the goal-independent based semantics this separation is made more explicit by defining a separate set of clauses that define these local components independently of any call descriptions. The same kind of separation can be done for the annotations: at each program point we can split the obtained description into a global component (the component that is due to the call description of the procedure) and a local component (the component that reflects the contribution to the exit description of the syntactical objects preceding the current program point). This is already partially done in the differential semantics, where the annotations are obtained by combining the current global component with the current local component. We make this separation explicit by pre-annotating all the program points with the goal-independent local components of the exit descriptions.

The reason why we insist so much on the annotations, and in this case even goal-independent annotation, is that such annotations form the link between the sequence of different analyses involved in the compile-time garbage collection system developed in this thesis. We must therefore rely on their correctness.

The extension is very natural, and one can easily prove that the obtained semantics is equivalent to the goal-independent based semantics, hence equivalent to the differential semantics of Mercury programs.

In this semantics we need a first set of clauses that define the goal-independent meaning of a rulebase, denoted by $\mathcal{S}em_{M\star p}$. This meaning consists of the goal-independent exit descriptions (like in $\mathcal{S}em_{M\star}$) and the goal-independent annotations. These annotations are collected in a table of type $\mathit{GIAnn} : \mathtt{pp} \rightarrow \mathit{Ans}$. The signatures of these clauses are given in Figure 5.12, while the clauses themselves are defined in Figure 5.13.

$$
\begin{array}{lcl}
\mathit{GIAnn} & = & \mathtt{pp} \rightarrow \mathit{Ans} \\
\mathbf{R}_{M\star p} & : & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
\mathbf{F}_{M\star p} & : & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & \equiv & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow \mathit{Atom} \rightarrow \mathit{Ans} \rightarrow \mathit{GIAnn} \\
\mathbf{Pr}_{M\star p} & : & \mathit{Procedure} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
\mathbf{G}_{M\star p} & : & \mathit{Goal} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \rightarrow \mathit{Ans} \rightarrow (\mathit{Ans} \times \mathit{GIAnn}) \\
\mathbf{L}_{M\star p} & : & \mathit{Literal} \rightarrow \mathit{GIProcMeaning} \rightarrow \mathit{Ans} \rightarrow \mathit{Ans}
\end{array}
$$

Figure 5.12: Signatures of the semantic functions used in $\mathcal{S}em_{M\star p}$.

Using the goal-independent meaning of the rulebase, we define the resulting goal-dependent semantics for Mercury programs, based on these goal-independent exit descriptions, and goal-independent program point annotations, the so called *pre-annotations*. Figure 5.14 shows the signatures of the functions defined in Figure 5.15. Note that the signature of $\mathbf{G}_{M\bullet p}$ differs from the signature of the semantics of goals given in our previous definitions: the semantics of a goal in this setting only consists of the adequate recording of the program point annotations. The exit descriptions as such are not required given the fact that the exit description of the procedure can be derived without the exit description of the goal in its body. Moreover, there is no need for defining the semantics of individual literals as the annotation itself can be done independent of the nature of the literal.

While the goal-dependent part shown in Figure 5.15 is defined in terms of the fixpoint operation, it can be shown that the goal-dependent derivation of the exit descriptions as well as the annotation of the program reaches a fixpoint after one single pass. Indeed, in order to compute the exit description of a procedure, the goal-independent information can simply be consulted, and all the program point annotations can be performed in one go using the call description and the already available goal-independent annotations.

$$
\begin{array}{lcl}
\mathbf{R}_{M\star p}[\![r]\!] & = & \mathrm{fix}(\mathbf{F}_{M\star p}[\![r]\!]) \\
\mathbf{F}_{M\star p}[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e,A)p_i(\overline{Y}) = & & \\
 & & \mathbf{Pr}[\![p_i]\!](e,A)p_i(\overline{Y}) \\
\mathbf{Pr}_{M\star p}[\![h \leftarrow g]\!](e,A)a & = & \mathrm{let}\ (\mathcal{S}_1,A_1) = \mathbf{G}_{M\star p}[\![g]\!](e,A)\mathsf{init}_{h\leftarrow g}\ \mathrm{in} \\
 & & \quad (\rho_{h\rightarrow a}\,((\mathcal{S}_1)|_h)\,,A_1) \\
\mathbf{G}_{M\star p}[\![g_1,g_2]\!](e,A)\mathcal{S} & = & \mathrm{let}\ (\mathcal{S}_1,A_1) = \mathbf{G}_{M\star p}[\![g_1]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \quad \mathbf{G}_{M\star p}[\![g_2]\!](e,A_1)\mathcal{S}_1 \\
\mathbf{G}_{M\star p}[\![g_1;g_2]\!](e,A)\mathcal{S} & = & \mathrm{let}\ (\mathcal{S}_1,A_1) = \mathbf{G}_{M\star p}[\![g_1]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \mathrm{let}\ (\mathcal{S}_2,A_2) = \mathbf{G}_{M\star p}[\![g_2]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \quad (\mathcal{S}_1 \sqcup \mathcal{S}_2, \mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_{M\star p}[\![\mathrm{if}\ g_1\ \mathrm{then}\ g_2\ \mathrm{else}\ g_3]\!](e,A)\mathcal{S} = & & \\
 & & \mathrm{let}\ (\mathcal{S}_1,A_1) = \mathbf{G}_{M\star p}[\![g_1]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \mathrm{let}\ (\mathcal{S}_2,A_2) = \mathbf{G}_{M\star p}[\![g_2]\!](e,A_1)\mathcal{S}_1\ \mathrm{in} \\
 & & \mathrm{let}\ (\mathcal{S}_3,A_3) = \mathbf{G}_{M\star p}[\![g_3]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \quad (\mathcal{S}_2 \sqcup \mathcal{S}_3, \mathsf{merge}(A_2,A_3)) \\
\mathbf{G}_{M\star p}[\![\mathrm{not}\ g]\!](e,A)\mathcal{S} & = & \mathrm{let}\ (\mathcal{S}_1,A_1) = \mathbf{G}_{M\star p}[\![g]\!](e,A)\mathcal{S}\ \mathrm{in} \\
 & & \quad (\mathcal{S},A_1) \\
\mathbf{G}_{M\star p}[\![l]\!](e,A)\mathcal{S} & = & (\mathbf{L}_{M\star p}[\![l]\!]\,e\,\mathcal{S}, A[\mathsf{pp}(l),\mathcal{S}]) \\
\mathbf{L}_{M\star p}[\![\mathrm{unif}]\!]\,e\,\mathcal{S} & = & \mathsf{add}(\mathrm{unif},\mathcal{S}) \\
\mathbf{L}_{M\star p}[\![p(\overline{X})]\!]\,e\,\mathcal{S} & = & \mathsf{comb}(\mathcal{S}, e(p(\overline{X})))
\end{array}
$$

Figure 5.13: Goal-independent semantics with pre-annotations $\mathcal{S}em_{M\star p}$

5.8.1 Implementation Issues

In analogy to $\mathcal{S}em_{M\star}$ and $\mathcal{S}em_{M\bullet}$, it is natural to implement the goal-independent part of the semantics in a bottom up way, while the goal-dependent part is still evaluated in a top-down way.

5.9 Overview of the different semantics

In the last few sections we have presented a number of different semantics for the Mercury programming language. The reason for developing these different semantics was to provide a link between the natural goal-dependent concrete interpretation of a Mercury program, and a goal-independent based abstract interpretation of such a program. We introduced the differential semantics as a necessary intermediate semantics that allowed us to clearly identify the conditions that need to be fulfilled by a description domain, in order to safely conclude that the semantics are equivalent. We have also added an additional semantics that pre-annotates the program with goal-independent program point annotations. These

$$
\begin{array}{lcl}
\mathbf{P}_{M\bullet p} & : & \mathit{Program} \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{R}_{M\bullet p} & : & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \\
\mathbf{F}_{M\bullet p} & : & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \\
 & \equiv & \mathit{RuleBase} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow \mathit{Atom} \times \mathit{Ans} \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{Pr}_{M\bullet p} & : & \mathit{Procedure} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \\
 & \equiv & \mathit{Procedure} \rightarrow (\mathit{GIProcMeaning} \times \mathit{GIAnn}) \\
 & & \quad \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \rightarrow \mathit{Atom} \times \mathit{Ans} \rightarrow (\mathit{Ans} \times \mathit{Ann}) \\
\mathbf{G}_{M\bullet p} & : & \mathit{Goal} \rightarrow \mathit{GIAnn} \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \\
 & & \quad \rightarrow \mathit{Ans} \rightarrow \mathit{Ann} \\
\mathbf{L}_{M\bullet p} & : & \mathit{Literal} \rightarrow \mathit{GIAnn} \rightarrow (\mathit{ProcMeaning} \times \mathit{Ann}) \\
 & & \quad \rightarrow \mathit{Ans} \rightarrow \mathit{Ann}
\end{array}
$$

Figure 5.14: Signatures of the semantic functions used in $\mathcal{S}em_{M\bullet p}$.

pre-annotations can be used to compute the goal-dependent annotations.

Schematically, we have the following relation between the semantics:

$$\mathcal{S}em_{M} \quad \overset{c}{\Longleftrightarrow}_{\text{(Th. 5.4)}} \quad \mathcal{S}em_{M\delta} \iff \mathcal{S}em_{M\bullet} \iff \mathcal{S}em_{M\bullet p}$$

where

- $\mathcal{S}em_{M}$: The natural goal-dependent semantics for Mercury. See Section 5.4.
- $\mathcal{S}em_{M\delta}$: The differential semantics for Mercury. See Section 5.6.
- $\mathcal{S}em_{M\bullet}$: The goal-independent based semantics for Mercury. See Section 5.7.
- $\mathcal{S}em_{M\bullet p}$: The goal-independent based semantics with pre-annotations for Mercury. See Section 5.8.

Hence, if a particular description domain D satisfies the conditions expressed in Theorem 5.4, then automatically, the results of interpreting a Mercury program in $\mathcal{S}em_{M}(D)$ are equivalent to the results of interpreting that program in $\mathcal{S}em_{M\bullet p}(D)$. Interestingly, the concrete domain of existentially quantified term equations $\wp(Eqn^{+})$ satisfies these conditions (Theorem 5.5). This means, that for an analysis in a description domain A defined in the context of the goal-independent based semantics with pre-annotations, it suffices that the auxiliary

$$
\begin{array}{lcl}
\mathbf{P}_{M\bullet p}[\![r;q]\!] & = & \text{let } (e^\star, A^\star) = \mathbf{R}_{M\star p}[\![r]\!] \text{ in} \\
 & & \quad \mathbf{G}_{M\bullet p}[\![q]\!]A^\star(\mathbf{R}_{M\bullet p}[\![r]\!](e^\star, A^\star)) \\
\mathbf{R}_{M\bullet p}[\![r]\!](e^\star, A^\star) & = & \text{fix}(\mathbf{F}_{M\bullet p}[\![r]\!](e^\star, A^\star)) \\
\mathbf{F}_{M\bullet p}[\![p_1 \ldots p_i \ldots p_{n_p}]\!](e^\star, A^\star)(e, A)p_i(\overline{Y})\mathcal{S} = & & \\
 & & \quad \mathbf{Pr}_{M\bullet p}[\![p_i]\!](e^\star, A^\star)(e, A)p_i(\overline{Y})\mathcal{S} \\
\mathbf{Pr}_{M\bullet p}[\![h \leftarrow g]\!](e^\star, A^\star)(e, A)a\mathcal{S} & = & \text{let } \mathcal{S}_g = \rho_{a\to h}\,(\,(\mathcal{S})|_a\,) \text{ in} \\
 & & \quad \text{let } A' = \mathbf{G}_{M\bullet p}[\![g]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad\quad (e^\star a, A') \\
\mathbf{G}_{M\bullet p}[\![g_1, g_2]\!]A^\star(e, A)\mathcal{S}_g & = & \text{let} A_1 = \mathbf{G}_{M\bullet p}[\![g_1]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad \mathbf{G}_{M\bullet p}[\![g_2]\!]A^\star(e, A_1)\mathcal{S}_g \\
\mathbf{G}_{M\bullet p}[\![g_1; g_2]\!]A^\star(e, A)\mathcal{S}_g & = & \text{let } A_1 = \mathbf{G}_{M\bullet p}[\![g_1]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad \text{let } A_2 = \mathbf{G}_{M\bullet p}[\![g_2]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad\quad \mathsf{merge}(A_1, A_2) \\
\mathbf{G}_{M\bullet p}[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!]A^\star(e, A)\mathcal{S}_g = & & \\
 & & \quad \text{let } A_1 = \mathbf{G}_{M\bullet p}[\![g_1]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad \text{let } A_2 = \mathbf{G}_{M\bullet p}[\![g_2]\!]A^\star(e, A_1)\mathcal{S}_g \text{ in} \\
 & & \quad \text{let } A_3 = \mathbf{G}_{M\bullet p}[\![g_3]\!]A^\star(e, A)\mathcal{S}_g \text{ in} \\
 & & \quad\quad \mathsf{merge}(A_2, A_3) \\
\mathbf{G}_{M\bullet p}[\![\text{not } g]\!]A^\star(e, A)\mathcal{S}_g & = & \mathbf{G}_{M\bullet p}[\![g]\!]A^\star(e, A)\mathcal{S}_g \\
\mathbf{G}_{M\bullet p}[\![l]\!]A^\star(e, A)\mathcal{S}_g & = & \text{let } \mathcal{S}_l = A^\star(\mathtt{pp}(l)) \text{ in} \\
 & & \quad\quad A[(\mathtt{pp}(l), \mathcal{S}_g), \mathsf{comb}(\mathcal{S}_g, \mathcal{S}_l))
\end{array}
$$

Figure 5.15: Goal-dependent semantics $\mathcal{S}em_{M\bullet p}$ based on the goal-independent semantics with pre-annotations $\mathcal{S}em_{M\star p}$.

functions init^A, add^A and comb^A safely approximate the concrete instantiations of these functions respectively, such that $\mathcal{S}em_{M\bullet p}(A)$ can be considered a safe approximation of $\mathcal{S}em_M(\wp(Eqn^+))$: $\mathcal{S}em_{M\bullet p}(A) \propto \mathcal{S}em_M(\wp(Eqn^+))$. No further conditions are required.

An interesting question to ask in the presence of the above presented semantics is how the abstract goal-dependent semantics of a Mercury program relates to the abstract goal-independent based semantics of the same program. If the abstract domain satisfies the equivalence conditions of Theorem 5.4, then, of course, the results obtained in both semantics are equally precise. In general, if some of the equivalence conditions are not satisfied by the abstract description domain, then the relative precision is not comparable. For some programs, the results obtained in one semantics can be more precise than the results obtained in the second semantics, but for other programs, this can be the other way around.

5.10 Mercury with Modules

In this chapter we have developed a number of semantics of Mercury programs, assuming that these Mercury programs are defined in one single module. Obviously, real Mercury programs will be split into different modules. The effect of compiling a program is to translate each of the modules of the program into the appropriate target code, and to combine these into one single executable. This means that for the concrete domain, the program can still be seen as one single monolithic block. Yet, for program analyses that operate on the source code, modules may pose a problem, especially if one wants to be able to analyse one module at a time, without having to load the source code of all of the modules of the user program. In such cases, pure goal-dependent analyses are not possible, leaving no other choice then to define goal-independent based analyses instead. Indeed, with goal-independent based analyses, the actual analysis is performed by the goal-independent part which can be used to give a meaning to each of the procedures defined in a module, *without* having to know the exact call descriptions of these procedures. The results of the goal-independent analysis, whether consisting of the goal-independent descriptions only, or if need be, also comprising the goal-independent annotations for (some of) the program points within the analysed procedures, may then be stored in dedicated files, the so called *optimisation interface files*. It then suffices to load these files during the analysis of other modules instead of having to load the complete source code.

Note that this scheme works fine as long as the modules are purely hierarchically organised, *i.e.*, the modules interdependence graph contains no loops. When loops are present, special techniques may be needed to correctly propagate the analysis results. One of these techniques consists of selectively recompiling modules (Bueno, García de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2001; Nethercote 2001), a technique that we will be using too, although in a less selective way (c.f. Chapter 11). If these loops are not only purely on the level of the modules, but also on the level of the procedures defined in them, then simple recompilation may not suffice. Indeed, if module m_1 defines a procedure p_1, and module m_2 defines a procedure p_2 such that p_1 is used in the procedure definition of p_2, and vice versa, then the general analysis scheme may have to be altered so as to guarantee precise yet correct results. See (Nethercote 2001) where some techniques handling such cases are detailed.

In this work, we consider looping modules, and especially looping procedure definitions in looping modules, as a bad programming habit reflecting a bad module structure of the user program. A simple tool reorganising the user code seems a much easier solution to this problem than adapting the analyses to correctly cope with those situations.

5.11 Conclusion

In this chapter we have presented a number of different definitions for the semantics of Mercury programs, the most important of which being the concrete goal-dependent semantics $\mathcal{S}em_M$ and the goal-independent based semantics with pre-annotations $\mathcal{S}em_{M\bullet p}$.

In the following chapters we define adequate new instantiations of $\mathcal{S}em_M$ reflecting not only the usual variable bindings (as in the particular instantiation $\mathcal{S}em_M(\wp(Eqn^+))$), but also other properties necessary to be able to reason about possible compile-time garbage collection. The semantics $\mathcal{S}em_{M\bullet p}$ and the differential semantics from which it is derived, will be used to formalise the analyses enabling us to approximate these run-time descriptions at compile-time.

Chapter 6

Data Structure Sharing

In this chapter we develop the required formalisations for the definition of a concrete domain suitable to represent terms that are shared in memory. This requires the extension of the classic concrete domain of variable bindings. We approximate this extended concrete domain by the abstract domain of data structure sharing. We use this domain to define the abstract sharing semantics of Mercury programs.

The concrete and abstract domains presented here for expressing data structure sharing are similar to the domains defined in (Mulkers 1991). The main difference lies in the use of these domains. Whereas Mulkers (1991) formalises structure sharing using the top-down goal-dependent abstract interpretation framework defined in (Bruynooghe 1991), we use our domain in the context of the different semantics defined for the Mercury language in the previous chapter. This requires a careful design of the concrete structure sharing domain with which we can prove that the results in a goal-independent setting are equivalent to the results obtained when using that domain in the natural semantics.

6.1 Motivation

Consider a type $\mathsf{t}_1 \in \Sigma_T$, defined by the type declaration:

$$\mathsf{t}_1 \rightarrow f(\mathsf{int}, \mathsf{int}); g(\mathsf{t}_1).$$

Let X and Y be of type t_1, and consider the following conjunction of construction unifications: `X <= f (1,2), Y <= g(X)`.

In the context of the Melbourne Mercury Compiler, this sequence of unifications instantiates two cells on the stack, and three heap cells. See Figure 6.1.

As the term pointed at by Y is constructed using X, it is natural that parts of the heap space are shared. More specifically, the term pointed at by X is shared

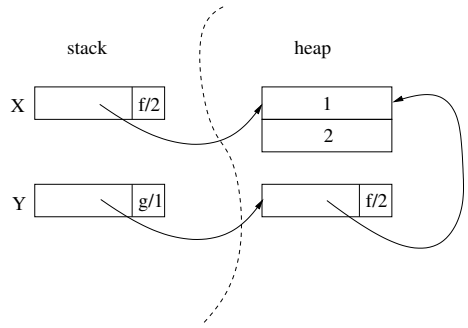


Figure 6.1: Graphical representation of memory sharing in the context of the Melbourne Mercury Compiler.

with the term to which the first argument of $g/1$ points to. This sharing information is essential in the process of detecting dead heap cells. Indeed, although X might not be used in the remainder of the code, as long as Y is used, the cells pointed at by X may not be removed from the heap, nor may they be reused for constructing other terms.

This small example illustrates our need to reason about parts of terms and their memory sharing in a formal setting.

6.2 Types, Terms, and Subterms

We start by associating so called *type trees* to each type defined in a given program. For these type trees, we define a formal way of selecting individual *type nodes*. This leads us to the notion of a *type selector*. Given the fact that each Mercury term is an instance of a type, we can directly map *type selectors* to *term selectors*. While the former select type nodes from the type tree of some type t, the latter designate subterms of a term of that type t. The formalisation of these subterms is used as a basis for the concrete domain of representing shared heap cells. Recursive types have infinite type trees. To be useful in the context of an abstract domain, we need a finite representation for such types. By introducing an equivalence relation on the selectors, we obtain the so called *type graphs*, hence guarantee a finite representation. Each partition under this equivalence relation represents a possibly infinite set of type nodes or subterms when applied on a specific type or term of that type respectively. This is the basis of the abstract domain that we use to represent structure sharing between the terms that the program variables may point at at run-time.

This section is based on (Vanhoof 2001) where the same notions are used for

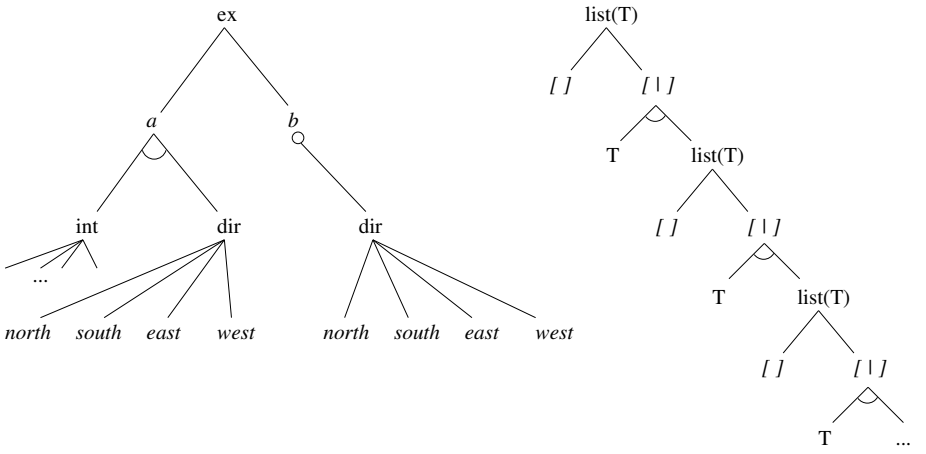


Figure 6.2: Type trees associated with the types ex and list(T) defined in Example 6.1. AND-nodes are marked in italic. If they have children, an arc (or circle) is used for connecting the edges to their children.

the binding time analysis that is developed in that work.

Recall that a type is a term from $\mathcal{T}(\Sigma_\mathcal{T}, \mathcal{V}_\mathcal{T})$, and that each type constructor in $\Sigma_\mathcal{T}$ is defined by a type declaration. A type declaration enumerates the different function symbols that are associated with a type, and also indicates the types of the arguments of these function symbols. As such, types can be represented by AND-OR trees. A type is an OR-node with a child node for each of the function symbols associated with that type by the type declaration. For each non-constant child, the type tree adds an AND-node representing all the types of the arguments the particular function symbol takes.

Example 6.1 *Consider the following type declarations:*

$$\begin{array}{lcl} \mathsf{dir} & \rightarrow & \mathit{north}; \mathit{south}; \mathit{east}; \mathit{west}. \\ \mathsf{ex} & \rightarrow & a(\mathsf{int}, \mathsf{dir}); b(\mathsf{dir}). \\ \mathsf{list}(\mathsf{T}) & \rightarrow & [\,]; [\mathsf{T} \mid \mathsf{list}(\mathsf{T})]. \end{array}$$

The type trees of ex *and* list(T) *are shown in Figure 6.2.*

The OR-nodes within a type tree are of particular interest as they represent the types of the subterms a term of that type may have. We refer to them as the *type nodes* of the type tree.

A type node of a type tree is uniquely identified by a path from the root of the type tree to the actual type node. Such a path is described as a sequence

from $\Sigma \times \mathbb{N}$. A path selecting a specific type node from a type tree is called a *type selector*. The empty sequence, called the *empty selector*, is denoted by ϵ. Applied to a type tree, ϵ selects the root node of the type tree, *i.e.*, the type itself. The sequence consisting of one single element (f, i), if applied to a type tree, selects the i'th type node of the function symbol f that can be selected from the OR-node at the root of the type tree. In general, a non-empty sequence $(f_1, i_1) \cdot (f_2, i_2) \cdot \ldots \cdot (f_n, i_n)$, if applied to a type tree, selects the type node that corresponds to the selector $(f_2, i_2) \cdot \ldots \cdot (f_n, i_n)$ if applied to the type node selected by the path (f_1, i_1) applied on the root of the original type tree.

Let *Selector* denote the set of all sequences over $\Sigma \times \mathbb{N}$. If $s_1, s_2 \in$ *Selector*, then $s_1 \bullet s_2$ represents the concatenation of these sequences, *i.e.*, $s_1 \bullet s_2 = (f_{1_1}, i_{1_1}) \cdot \ldots \cdot (f_{n_1}, i_{n_1}) \cdot (f_{1_2}, i_{1_2}) \cdot \ldots \cdot (f_{n_2}, i_{n_2})$ if $s_1 = (f_{1_1}, i_{1_1}) \cdot \ldots \cdot (f_{n_1}, i_{n_1})$ and $s_2 = (f_{1_2}, i_{1_2}) \cdot \ldots \cdot (f_{n_2}, i_{n_2})$. The empty selector ϵ is a neutral element for the concatenation operation: $\forall s \in$ *Selector* : $\epsilon \bullet s = s \bullet \epsilon = s$.

The formal definition of a type tree is given in terms of these type selectors:

Definition 6.1 (Type tree) *Given a type* $\mathsf{t} \in \mathcal{T}(\Sigma_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}})$*, the* type tree *of* t*, denoted by* $\mathcal{TT}_{\mathsf{t}}$*, is the set of type selectors from Selector – the set of all sequences over* $\Sigma \times \mathbb{N}$*, where:*

- $\epsilon \in \mathcal{TT}_{\mathsf{t}}$;
- *if* $\mathsf{t} = h(\mathsf{T}_1, \ldots, \mathsf{T}_n)\theta$ *where* h/n *is defined by the type description*

$$h(\mathsf{T}_1, \ldots, \mathsf{T}_n) \rightarrow f_1(\overline{\mathsf{t}_1}); f_2(\overline{\mathsf{t}_2}); \ldots f_i(\overline{\mathsf{t}_i})$$

  and where θ *is a type substitution, then for all functors* f_j/m *with* $j \in \{1, \ldots, i\}$*, and for each argument position* $k \in \{1, \ldots, n\}$ *we have* $(f_j, k) \bullet s \in \mathcal{TT}_{\mathsf{t}}$*, where* $s \in \mathcal{TT}_{(\mathsf{t}')\theta}$ *with* $\mathsf{t}' = \mathsf{t}^{(f_j,k)}$.

Given a type t, and $s \in$ *Selector* a type selector in t's type tree, $s \in \mathcal{TT}_{\mathsf{t}}$, then we use the notation t^s to denote the type of the type node identified by the type selector s in the type tree for t.

Definition 6.2 (Ancestor Type Node) *Let* s_1 *and* s_2 *be two type selectors in the type tree of a type* t, i.e., $s_1, s_2 \in \mathcal{TT}_{\mathsf{t}}$*. The type node selected by* s_1 *in the type tree of* t *is said to be an* ancestor type node *of the type node selected by* s_2 *in the type tree of* t*, if and only if there exists a selector* s*, such that* $s_1 \bullet s = s_2$.

Note that for every type t and every selector $s \in \mathcal{TT}_{\mathsf{t}}$, ϵ is always an ancestor of s.

Example 6.2 *Using the type trees depicted in Figure 6.2, let* $s_1 = ([\,|\,], 1)$ *and* $s_2 = ([\,|\,], 2)$*, then we have:*

$$\begin{array}{lcl} \mathsf{ex}^{\epsilon} & = & \mathsf{ex} \\ s_2 \bullet s_1 & = & ([\,|\,], 2) \cdot ([\,|\,], 1) \\ \mathsf{list}(\mathsf{T})^{s_1} & = & \mathsf{T} \\ \mathsf{list}(\mathsf{T})^{s_2} & = & \mathsf{list}(\mathsf{T}) \\ \mathsf{list}(\mathsf{T})^{s_2 \bullet s_1} & = & \mathsf{T} \end{array}$$

Example 6.3 *If the types* dir, ex *and* list(T) *are as defined in Example 6.1, then their type trees are:*

$$\begin{array}{lcl} \mathcal{TT}_{\mathsf{dir}} & = & \{\epsilon\} \\ \mathcal{TT}_{\mathsf{ex}} & = & \{\epsilon, (a,1), (a,2), (b,1)\} \\ \mathcal{TT}_{\mathsf{list}(\mathsf{T})} & = & \left\{ \begin{array}{l} \epsilon, \\ ([\,|\,], 1), \\ ([\,|\,], 2), \\ ([\,|\,], 2) \cdot ([\,|\,], 1), \\ ([\,|\,], 2) \cdot ([\,|\,], 2), \\ ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 1), \\ ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 2), \\ ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 1), \\ ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 2) \cdot ([\,|\,], 2), \\ \ldots \end{array} \right\} \end{array}$$

Mercury is a strongly typed language. This means that every term used in a program can be given a type. Let τ be a term of type t, then this term can be seen as a subset of the type tree of t, such that in each encountered OR-node, all but one branch is pruned away.

Definition 6.3 (Term Tree, Term Selectors) *Let* τ *be a term of type* t*, then the* term tree $\mathcal{T}_\tau \subseteq \mathcal{TT}_{\mathsf{t}}$ *denotes the set of selectors that correctly select any of the type nodes in the graphical representation of* τ*. Elements from* $\mathcal{T}_\tau$ *are called* term selectors*.*

Example 6.4 *Consider the term* $a(3, north)$*. This term is of type* ex*, and can be graphically represented by a tree. Figure 6.3 shows this tree, and its explicit mapping to the type tree. Here* $\mathcal{T}_{a(3,north)} = \{\epsilon, (a,1), (a,2)\}$*.*

We add the notion of compatible term selectors:

Definition 6.4 (Compatible Selectors) *Given a type* t*, and selectors* $s_1, s_2 \in \mathcal{TT}_{\mathsf{t}}$*, then* s_1 *and* s_2 *are said to be* compatible*, denoted by* $s_1 \bowtie s_2$*, iff:*

- *either* $s_1 = (f, n_1) \bullet s_1'$, $s_2 = (f, n_2) \bullet s_2'$, $n_1 \neq n_2$*;*

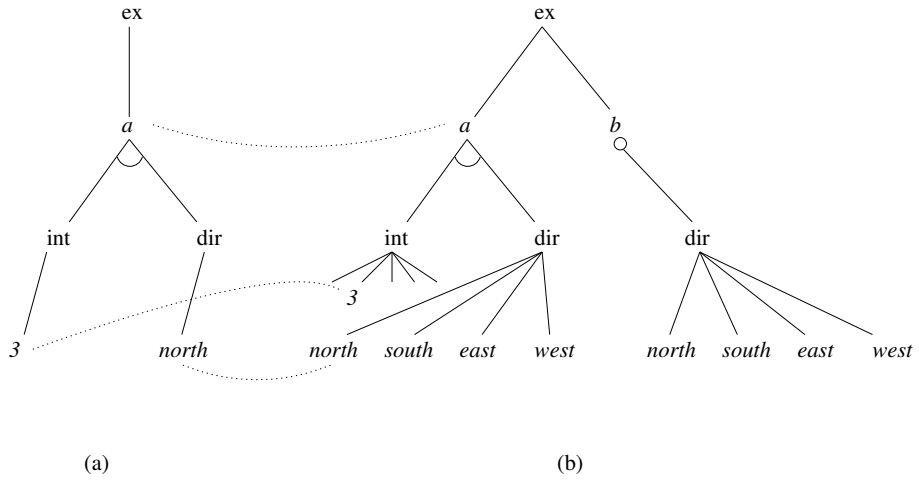


Figure 6.3: Schematic representation of the term $a(3, north)$ of type ex (a) and its mapping on the full type tree of ex (b).

- *or* $s_1 = s \bullet s'_1$, $s_2 = s \bullet s'_2$, $s'_1 \bowtie s'_2$.

Example 6.5 *Consider the type* ex *as defined in Example 6.1. The selectors* $(a, 1)$ *and* $(a, 2)$ *are compatible because they select type nodes starting from the same* AND*-node. The selectors* $(a, 1)$ *and* $(b, 1)$ *are not compatible.*

We have the following interesting property:

Proposition 6.1 *Given a term* τ *of type* t*, then all term selectors in* $\mathcal{T}_\tau$ *are mutually compatible.*

Definition 6.5 (Valid Term Selector) *Given a term* τ *and a selector* s *then* s *is a* valid *selector for* τ *iff* $s \in \mathcal{T}_\tau$.

Definition 6.6 (Subterm) *Let* τ *be a term of type* t*, and* $s \in \mathcal{T}_\tau$*, then* s *applied on the term* τ*, denoted by* τ^s*, selects the* subterm *with term tree* $\mathcal{T}_{\tau^s} = \{s' | s \bullet s' \in \mathcal{T}_\tau\}$*. The type of this subterm is given by* t^s.

The elements of type trees and term trees are called *type selectors* and *term selectors* respectively. In general, we use "*selector*" to refer to either of these selectors.

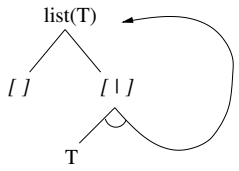


Figure 6.4: Graphical representation of the type graph for $\mathsf{list}(\mathsf{T})$.

Example 6.6 *Let* $\tau_1 = a(3, north)$ *of type* ex. *Let* $s_1 = (a, 1)$, $s_2 = (a, 2)$ *and* $s_3 = (b, 1)$. *Let* $\tau_2 = [3, 4, 5]$ *of type* $\mathsf{list}(\mathsf{int})$[1] *and* $s_a = \epsilon$, $s_b = ([\,|\,], 2)$, *and* $s_c = ([\,|\,], 2) \cdot ([\,|\,], 1)$. *Then*

$$\begin{array}{lcl} \mathcal{T}_{a(3,north)} & = & \{\epsilon, (a,1), (a,2)\} \\ \mathcal{T}_{[3,4,5]} & = & \{\epsilon, ([|],1), ([|],2), \\ & & ([|],2)\cdot([|],1), ([|],2)\cdot([|],2), \\ & & ([|],2)\cdot([|],2)\cdot([|],1), ([|],2)\cdot([|],2)\cdot([|],2)\} \end{array}$$

$$\begin{array}{lcllcl} \tau_1^{s_1} & = & 3 & \tau_2^{s_a} & = & [3,4,5] \\ \tau_1^{s_2} & = & north & \tau_2^{s_b} & = & [4,5] \\ & & & \tau_2^{s_c} & = & 4 \end{array}$$

Note that for example, s_3 *is not a valid selector for* τ_1.

Type trees can in theory be infinite. The type tree for the type $\mathsf{list}(\mathsf{T})$ (Figure 6.2 and Example 6.3) illustrates this. A finite representation can be ensured by using *type graphs* instead. A type graph is obtained from a type tree by folding a branch tree leading to a type node back to an ancestor type node of the same type, if such an ancestor type node exists.

Example 6.7 *Figure 6.4 shows the type graph for the* $\mathsf{list}(\mathsf{T})$*-type.*

We formalise the notion of type graphs through an equivalence relation defined on the set of selectors.

Definition 6.7 (Selector equivalence) *Two selectors are considered to be* equivalent *for a type* t *if the type nodes that they select have the same type and if either one of the selectors is an ancestor of the other one. Formally: Let* $s_1, s_2 \in$ *Selector applicable to a type* t. *If* $\mathsf{t}^{s_1} = \mathsf{t}^{s_2}$ *and if* $\exists s \in$ *Selector* : $s_1 \bullet s = s_2$ *or* $s_2 \bullet s = s_1$, *then* $s_1 \equiv s_2$.

As $\equiv$ is reflexive, symmetric and transitive, it is a valid equivalence relation.

It is worthwhile to note that having the same type is not enough for two type nodes to be equivalent.

[1] Note that τ_2 is of type $\mathsf{list}(\mathsf{T})\theta$ with the type substitution $\theta = \{\mathsf{T}/\mathsf{int}\}$.

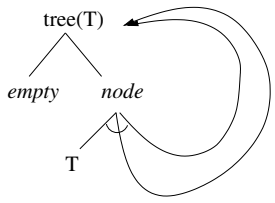


Figure 6.5: Graphical representation of the type graph for $\mathsf{tree}(\mathsf{T})$ as defined in Example 6.9.

Example 6.8 *Consider a type* $\mathsf{pair}(\mathsf{T})$ *defined by the following type description:*

$$\mathsf{pair}(\mathsf{T}) \rightarrow (\mathsf{T} - \mathsf{T})$$

Although the selectors $((-), 1)$ *and* $((-), 2)$ *select type nodes of the same type, they are not considered equivalent:* $((-), 1) \not\equiv ((-), 2)$.

Example 6.9 *Consider the type* $\mathsf{tree}(\mathsf{T})$ *defined by the following type description:*

$$\mathsf{tree}(\mathsf{T}) \rightarrow \mathit{empty}\,;\, \mathit{node}(\mathsf{T}, \mathsf{tree}(\mathsf{T}), \mathsf{tree}(\mathsf{T}))$$

for which the type graph is depicted in Figure 6.5. Here, the selectors $(\mathit{node}, 2)$, $(\mathit{node}, 3)$ *are equivalent to the root node, i.e. to the selector* ϵ*, and therefore are equivalent to each other. This illustrates the idea that two selectors can be equivalent, although they are not an ancestor of each other.*

An interesting property of type selector equivalence is the following one that states that if two selectors of a type are equivalent, then they must have a common ancestor with which they are equivalent.

Proposition 6.2 *Given a type* t *with type tree* $\mathcal{TT}_{\mathsf{t}}$*, and* $s_1, s_2 \in \mathcal{TT}_{\mathsf{t}}$*. If* $s_1 \equiv s_2$ *then* $\exists s \in \mathcal{TT}_{\mathsf{t}}$ *such that* $\exists e_1, e_2 \in$ *Selector for which* $s_1 = s \bullet e_1$ *and* $s_2 = s \bullet e_2$ *and* $s_1 \equiv s_2 \equiv s$.

Note that in so called *type based* analyses (Bruynooghe, Codish, Gallagher, Genaim, and Vanhoof 2003; Lagoon, Mesnard, and Stuckey 2003), the equivalence relation between selectors is based on type equivalence only and is therefore a coarser definition of equivalence. The effect this has on the precision of the resulting analysis and how it compares to the precision obtained with our current definition of selector equivalence is discussed in Section 12.7.

In this thesis we only consider types for which the set of types of the type nodes that can be selected is finite. This means that types may not be defined in

terms of strict instances of themselves. A type $\mathsf{badlist}(\mathsf{T})$ defined by the following description rule is therefore excluded:

$$\mathsf{badlist}(\mathsf{T}) \rightarrow [\,]; [\mathsf{T} \mid \mathsf{badlist}(\mathsf{badlist}(\mathsf{T}))]$$

The restriction allows us to guarantee that the equivalence relation $\equiv$ partitions each possibly infinite type tree into a finite number of equivalence classes. This is a common restriction, although see (Okasaki 1998) for the use of such types.

Finally, for a type t and $s \in \mathcal{TT}_{\mathsf{t}}$, we use the notations $[s]_{\mathsf{t}}$ and $\overline{s}_{\mathsf{t}}$ to denote the equivalence class of selector s and the minimal element of that equivalence class respectively in the context of that type t. Formally:

$$\begin{array}{lcl} [s]_{\mathsf{t}} & = & \{s' \in \mathcal{TT}_{\mathsf{t}} \mid s' \equiv s\} \\ \overline{s}_{\mathsf{t}} & = & s' \in [s]_{\mathsf{t}} \text{ such that } \forall s'' \in [s]_{\mathsf{t}} : s'' = s' \bullet e, \text{ for some } e \in \mathit{Selector}. \end{array}$$

If the type context is clear, $[s]_{\mathsf{t}}$ and $\overline{s}_{\mathsf{t}}$ are abbreviated to $[s]$ and $\overline{s}$ respectively.

Example 6.10 *In the context of the type* $\mathsf{list}(\mathsf{T})$*, we have*

$$\overline{\epsilon} = \overline{([|], 2)} = \overline{([|], 2) \cdot ([|], 2)} = \ldots = \epsilon$$

and

$$\overline{([|], 1)} = \overline{([|], 2) \cdot ([|], 1)} = \overline{([|], 2) \cdot ([|], 2) \cdot ([|], 1)} = \ldots = ([|], 1)$$

Definition 6.8 (Type Graph) *Given a type* $\mathsf{t} \in \mathcal{T}(\Sigma_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}})$*, the* type graph *of* t*, denoted by* $\mathcal{TG}_{\mathsf{t}}$*, is the set of minimal elements of the equivalence classes that are obtained using* $\equiv$ *as the equivalence relation over the type tree* $\mathcal{TT}_{\mathsf{t}}$ *of that type:*

$$\mathcal{TG}_{\mathsf{t}} = \{\overline{s} \mid s \in \mathcal{TT}_{\mathsf{t}}\}$$

The elements of $\mathcal{TT}_{\mathsf{t}}$ and $\mathcal{TG}_{\mathsf{t}}$ are both in *Selector*, therefore, to make a clear distinction between them, we systematically write selectors of the type graph of a type by over-lining them.

Example 6.11 *The type graphs of the types defined in Example 6.1 are:*

$$\begin{array}{lcl} \mathcal{TG}_{\mathsf{dir}} & = & \{\overline{\epsilon}\} \\ \mathcal{TG}_{\mathsf{ex}} & = & \{\overline{\epsilon}, \overline{(a,1)}, \overline{(a,2)}, \overline{(b,1)}\} \\ \mathcal{TG}_{\mathsf{list(T)}} & = & \{\overline{\epsilon}, \overline{([\,|\,], 1)}\} \end{array}$$

Example 6.12 *Consider the types* $\mathsf{list}(\mathsf{A})$ *and* $\mathsf{listone}(\mathsf{B})$ *defined by the following type declarations:*

$$\begin{array}{lcl} \mathsf{list}(\mathsf{A}) & \rightarrow & [\,]; [\mathsf{A} | \mathsf{list}(\mathsf{A})]. \\ \mathsf{listone}(\mathsf{B}) & \rightarrow & [\mathsf{B} | \mathsf{list}(\mathsf{B})]. \end{array}$$

At first sight it might seem that the type $\mathsf{listone(B)}$ *defines all the lists with at least one element. In that case, we would have* $\mathsf{listone(B)}^{\epsilon} = \mathsf{list(B)}$, *and* $\overline{\epsilon} = \overline{([|],2)} = \overline{([|],2)\cdot([|],2)} = \epsilon$ *for the type* $\mathsf{listone(B)}$. *Also* $\mathcal{TG}_{\mathsf{listone(B)}} = \{\epsilon, ([|],1)\}$.

This is wrong. Indeed, although both types use the same term constructor $[|]$, *they are different. This becomes clear by subscribing the term constructor from* $\mathsf{listone(B)}$ *with* lo, *thus* $[|]_{lo}$. *We then have:*

$$\mathcal{TG}_{\mathsf{listone(B)}} = \{\overline{\epsilon}, \overline{([|]_{lo},1)}, \overline{([|]_{lo},2)}, \overline{([|]_{lo},2)\cdot([|],1)}\}$$

While a single selector $s \in \mathcal{TT}_{\mathsf{t}}$, applied on a term τ of type t, selects one specific subterm τ^s of that term (if it exists), we consider that an element $\overline{s} \in \mathcal{TG}_{\mathsf{t}}$ applied to τ designates the set of subterms selected by the selectors of the equivalence class $[s]$.

Example 6.13 *Let* $\tau = [3,4,5]$, *then*

$$\begin{array}{lcl} \tau^{\overline{\epsilon}} & = & \{[3,4,5],[4,5],[5],[\,]\} \\ \tau^{\overline{([|],1)}} & = & \{3,4,5\} \end{array}$$

The first selector, $\overline{\epsilon}$, *selects all the sublists of the original list, including the original list itself. The second selector, applied to a list, selects all the elements within that list.*

If X is a variable bound to a term τ, and s a valid selector for τ, then we use the notation for applying selectors on terms also in the context of variables pointing to these terms. Thus X^{ϵ} denotes the term τ, and X^s the subterm τ^s. Similarly, if $\overline{s} \in \mathcal{TG}_{\mathtt{type}(\tau)}$, then $X^{\overline{s}}$ denotes the set of subterms selected by the elements from the equivalence class $[s]$.

6.3 Concrete Domain for Structure Sharing

In Chapter 5 we introduced the concrete semantics of our Mercury language using the concrete domain of variable bindings represented by existentially quantified term equations. This concrete domain is detailed enough to keep track of the variable bindings, *i.e.*, the values to which the variables in use are bound. Yet as such it can not be used to reflect the more low-level information of memory sharing that is needed in our work. We therefore augment this concrete domain with a domain that does keep track of the sharing that may exist between terms during the execution of the program.

To simplify the presentation, we do not make a distinction between primitive types and other types and consider that terms of primitive types may also be shared in memory[2].

[2]Recall that in the MMC terms of primitive types have a simplified representation and can therefore not be shared, c.f. Section 3.8.3.

6.3.1 From Data Structure to Collecting Sharing Sets

We use the notion of *data structure* to refer to the physical memory required for storing the representation of a term (or subterm) referred to from the user program. In Mercury data structures are memory cells from the heap.

Definition 6.9 (Data structure) *Consider a variable X bound to a term τ at a specific moment during the execution of a program – described by the ex-equation e, then $\langle e, X^s \rangle$ represents the heap cells used to store the subterm τ^s. The tuple $\langle e, X^s \rangle$ is called the* data structure associated with the term X^s in the context *e. We use X^s, without an explicit context, to denote so called* context-free data structures.

Type information is not explicitly present in the definition of a data structure as we assume that a data structure is used in the context of a specific procedure in which we consider the type information of its variables to be implicitly present (c.f. Section 4.2.2).

Let X be a variable of type $\mathtt{type}(X)$, then we use $\mathcal{D}_X$ to denote the set of context-free data structures of the terms (and subterms of these terms) that X may point to: $\mathcal{D}_X = \{X^s \mid s \in \mathcal{TT}_{\mathtt{type}(X)}\}$. Let VI be the set of variables of interest. We generalise the above notation such that $\mathcal{D}_{VI}$ denotes the context-free data structures over all the variables of interest: $\mathcal{D}_{VI} = \{X^s \mid X^s \in \mathcal{D}_X, X \in VI\}$. The set of data structures over all the variables of interest is the domain $\langle Eqn^+, \mathcal{D}_{VI} \rangle$.

Definition 6.10 (Valid Data Structure) *A data structure $\langle e, X^s \rangle \in \langle Eqn^+, \mathcal{D}_{VI} \rangle$ is* valid *iff $\exists \tau \in \mathcal{T}(\mathcal{V}, \Sigma) : e \models X = \tau$ and $s \in \mathcal{T}_\tau$.*

Note that context-free data structures are neither valid nor invalid as there is no context to validate or invalidate them in.

By abuse of notation we use X^s to refer to the term X^s as well as to the (context-free) data structure associated with that term. In that case we also say "the data structure X^s" instead of "the data structure associated with the term X^s".

We say that two terms are *shared* if the memory used to store these terms is the same.

Definition 6.11 (Sharing Pair) *Let X^{s_X} and Y^{s_Y} be two data structures in a context $e \in Eqn^+$, then X^{s_X} and Y^{s_Y} are* shared *iff the terms X^{s_X} and Y^{s_Y} occupy the same memory space in the environment e, thus $X^{s_X} = Y^{s_Y}$. This memory constraint in the context of the variable bindings e is denoted by $\langle e, (X^{s_X} - Y^{s_Y}) \rangle$, and is called a* sharing pair. *In analogy with data structures, we call $(X^{s_X} - Y^{s_Y})$ a* context-free *sharing pair if used in the absence of an explicit ex-equation.*

We use $\mathcal{SD}_{VI}$ to denote the set of context-free sharing pairs over a set of variables VI.

Of course, if two terms share memory then these terms must be equal and therefore also have the same type:

Corollary 6.1 *If* $\langle e, (X^{s_X} - Y^{s_Y})\rangle$ *is a sharing pair, and* $e \models X = \tau_X \wedge Y = \tau_Y$*, then* $\tau_X^{s_X} = \tau_Y^{s_Y}$*, and* $\mathtt{type}(X^{s_X}) = \mathtt{type}(Y^{s_Y})$.

Obviously, sharing is commutative: $\langle e, (X^{s_X} - Y^{s_Y})\rangle \Leftrightarrow \langle e, (Y^{s_Y} - X^{s_X})\rangle$.

Definition 6.12 (Valid Sharing Pair) *A sharing pair* $\langle e, (X^{s_X} - Y^{s_Y})\rangle$ *is* valid *iff the data structures* $\langle e, X^{s_X}\rangle$ *and* $\langle e, Y^{s_Y}\rangle$ *are valid, and of course,* X^{s_X} *and* Y^{s_Y} *share.*

If an environment e has more then one sharing relation, then we use the notion of a *sharing set*:

Definition 6.13 (Sharing Set) *Let* $e \in Eqn^+$ *and* $C \in \wp(\mathcal{SD}_{VI})$*, then* $\langle e, C\rangle$ *is called a* sharing set *with sharing pairs* C *in the context* e*. If* C *is used without an explicit context, then it is called a* context-free sharing set.

Again, we define the notion of validity:

Definition 6.14 (Valid Sharing Set) *Let* $\langle e, C\rangle$ *be a sharing set, then this sharing set is* valid *iff* $\forall S \in C : \langle e, S\rangle$ *is valid.*

If a sharing set is valid, then all data structures used to express the sharing information are valid too. This means that if a variable X is involved in the sharing set, then all selectors used on X must belong to the same term tree, hence, must be compatible (c.f. Proposition 6.1). We therefore have the following corollary:

Corollary 6.2 *If* $\langle e, C\rangle \in \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ *is a valid sharing set, then all term selectors used for the same variable in* C *are mutually compatible:*

$$\forall X \in VI : (X^{s_X} - Y^{s_Y}), \left(X^{s'_X} - Z^{s_Z}\right) \in C \Rightarrow s_X \bowtie s'_X$$

Example 6.14 *Consider the type declarations*

$$\begin{array}{lcl} \mathsf{t1} & \rightarrow & f(\mathsf{t2}); g(\mathsf{t2}). \\ \mathsf{t2} & \rightarrow & h(\mathsf{int}). \end{array}$$

and three variables X*,* Y *and* Z *of type* t1*. The context-free sharing set*

$$\left\{\left(X^{(f,1)} - Y^{(f,1)}\right), \left(X^{(g,1)} - Z^{(f,1)}\right)\right\}$$

can not be valid in any environment $e \in Eqn^+$*. Indeed,* $\nexists \tau \in \mathcal{T}(\mathcal{V}, \Sigma)$ *such that* $(f,1) \in \mathcal{T}_\tau$ *and* $(g,1) \in \mathcal{T}_\tau$ *as this would indicate that the variable* X *is bound to a term with outermost functor* $f/1$*, but also to a term with outermost functor* $g/1$*. These two bindings are incompatible.*

Mercury is a logic programming language, hence our domain needs to be suitable to express multiple solutions and multiple inputs. We therefore need sets of sharing sets to describe all variable bindings and their sharing.

Definition 6.15 (Collecting Sharing Set) *A set of sharing sets*

$$\{\langle e_1, C_1\rangle, \ldots, \langle e_n, C_n\rangle\} \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$$

is called a collecting sharing set.

Definition 6.16 (Valid Collecting Sharing Set) *A collecting sharing set ECS is valid if each of the sharing sets $EC \in ECS$ is valid.*

We limit our domain to valid sharing sets only. Therefore, the concrete domain that we shall use to represent sharing between terms in a given program will be the domain of valid collecting sharing sets, by abuse of notation also denoted with $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, where VI is the set of variables of interest.

6.3.2 Operations

We define some operations on the domain of sharing sets $\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ and extend them to the actual concrete sharing domain $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$.

When two data structures share, then their subterms share too. We make this explicit using the termshift closure operation defined below.

Definition 6.17 (Termshift) *The* termshift*-operation expands a sharing set to a new set that explicitly includes the sharing information between all the subterms of the sharing data structures included in the initial sharing set. Formally, we have* termshift $: \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle \to \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ *such that:*

$$\mathsf{termshift}(\langle e, C\rangle) \quad = \quad \langle e, C'\rangle$$

where

$$C' \quad = \quad \left\{ (X^{s_X \bullet s} - Y^{s_Y \bullet s}) \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \in C, \\ e \models X = \tau_X, \\ s_X \bullet s \in \mathcal{T}_{\tau_X} \end{array} \right\}$$

As the term trees of the terms are finite sets, C' is guaranteed to be a finite set too.

Note that as $(X^{s_X} - Y^{s_Y}) \in C$, if $e \models X = \tau_X$, then there must be a τ_Y such that $\tau_X^{s_X} = \tau_Y^{s_Y}$, which implies that if $s_X \bullet s \in \mathcal{T}_{\tau_X}$, then $s_Y \bullet s \in \mathcal{T}_{\tau_Y}$ (Corollary 6.1).

We define the termshift operation for context-free structure sharing sets.

Definition 6.18 (Termshift for context-free structure sharing) *Let C be a context-free structure sharing set,* i.e., *$C \in \wp(\mathcal{SD}_{VI})$, then*

$$\mathsf{termshift}(C) \quad = \quad \left\{ (X^{s_X \bullet s} - Y^{s_Y \bullet s}) \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \in C, \\ s_X \bullet s \in \mathcal{TT}_{\mathtt{type}(X)} \end{array} \right\}$$

Note that in this definition the absence of a concrete environment is compensated by using the type tree of the variable instead of the term tree of the term to which that variable would have been bound in a given environment. This definition may lead to an infinite set of structure sharing pairs in the resulting set. This is not a problem if we consider the termshift operation to be a *lazy* termshift (Bruynooghe, Janssens, and Kågedal 1997) where terms are termshifted on demand.

When a data structure X^{s_X} shares with Y^{s_Y}, but also with Z^{s_Z}, then automatically this means that Y^{s_Y} is shared with Z^{s_Z}. In other words, sharing is a transitive property between data structures. To compute all the shared data structures implied by transitivity over a set of data structures we use the transitive closure operation, transclos. The following definition gives a general definition of transclos and can be adapted to the sharing sets in $\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ in a straightforward way.

Definition 6.19 (Transitive Closure) *Consider a set of elements A. Then the transitive closure* $\mathsf{transclos} : A \times A \to A \times A$ *is defined as the minimal relation such that if (a, b) and (b, c) is in the transitive closure of a set S, then also (a, c) must be in S.*

For elements in $\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ we define:

$$\mathsf{transclos}(\langle e, C\rangle) = \langle e, \mathsf{transclos}(C)\rangle$$

Definition 6.20 (Projection) *Let $\langle e, C\rangle \in \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ and $V \subseteq VI$, then the projection of $\langle e, C\rangle$ on the variables V is defined as:*

$$(\langle e, C\rangle)|_V \quad = \quad \langle\, (e)|_V \,,\, (C)|_V \rangle$$

where $(e)|_V$ is the projection operation defined on Eqn^+ (Definition 2.8), and $(C)|_V$ is given by:

$$(C)|_V \quad = \quad \left\{ (X^{s_X} - Y^{s_Y}) \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \in \mathsf{transclos}(\mathsf{termshift}(\langle e, C\rangle)) \\ \textit{and} \\ X, Y \in V \end{array} \right\}$$

Definition 6.21 (Renaming) *Let $\langle e, C\rangle \in \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ and $\overline{X}, \overline{Y}$ be two sequences of variables in VI, then the renaming of $\langle e, C\rangle$ w.r.t. to the mapping $\overline{X} \to \overline{Y}$ is defined as:*

$$\rho_{\overline{X}\to\overline{Y}}\,(\langle e, C\rangle)) = \langle \rho_{\overline{X}\to\overline{Y}}\,(e)\,, \rho_{\overline{X}\to\overline{Y}}\,(C)\rangle$$

where $\rho_{\overline{X}\to\overline{Y}}\,(e)$ is given by Definition 2.9, and $\rho_{\overline{X}\to\overline{Y}}\,(C)$ is:

$$\rho_{\overline{X}\to\overline{Y}}\,(C) \quad = \quad \left\{ \left({X_i}^{s_{X_i}} - {X_j}^{s_{X_j}} \right) \;\middle|\; \begin{array}{l} \left({X_i}^{s_{X_i}} - {X_j}^{s_{X_j}} \right) \in C, \\ (X_i, Y_i), (X_j, Y_j) \in \overline{X} \to \overline{Y} \} \end{array} \right\}$$

We generalise these operations to elements from $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$. Let $ECS \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, then

$$\begin{array}{lcl} \mathsf{termshift}(ECS) & = & \{\mathsf{termshift}(EC) \mid EC \in ECS\} \\ \mathsf{transclos}(ECS) & = & \{\mathsf{transclos}(EC) \mid EC \in ECS\} \\ (ECS)|_V & = & \{\,(EC)|_V \mid EC \in ECS\} \\ \rho_{\overline{X}\to\overline{Y}}(ECS) & = & \{\rho_{\overline{X}\to\overline{Y}}(EC) \mid EC \in ECS\} \end{array}$$

6.3.3 Ordering

The sharing domain is a domain composed of the traditional information of variable bindings augmented with an explicit memory sharing part. It is therefore natural to define an ordering in terms of the ordering of each of the components of the domain.

We start by ordering sharing sets.

We define the following order relation for context-free structure sharing sets.

Definition 6.22 (Ordering in $\wp(\mathcal{SD}_{VI})$) *Let $a \in \mathcal{SD}_{VI}$ be a context-free sharing pair and $C, C_1, C_2 \in \wp(\mathcal{SD}_{VI})$ context-free sharing sets.*

We say that a is directly subsumed *by the sharing set C iff $a \in C$, where $\in$ is the usual set-inclusion. The sharing pair a is* subsumed *by the sharing set C, denoted by $a \preceq_c C$, iff a is directly subsumed by* $\mathsf{transclos}(\mathsf{termshift}(C))$.

The sharing set C_1 is subsumed *by C_2, denoted by $C_1 \subseteq_c C_2$, iff each of the sharing pairs expressed in C_1 is subsumed by C_2: $\forall a \in C_1 : a \preceq_c C_2$.*

The least upper bound operation for context-free structure sharing sets is simply their union.

Two context-free sharing sets are *equivalent* if they are mutually subsumed, *i.e.*, $\forall C_1, C_2 \in \wp(\mathcal{SD}_{VI})$, C_1 is equivalent with C_2, simply denoted as $C_1 \equiv C_2$, iff $C_1 \subseteq_c C_2$ and $C_2 \subseteq_c C_1$.

In $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, we have:

Definition 6.23 (Ordering in $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$) *Collecting sharing sets are ordered by the set-inclusion operation, modulo the equivalence of each of the components.*

Let $EC \in \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$ and $ECS \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, then the sharing set EC is subsumed *by ECS, denoted with $EC \leq_c ECS$, iff $\exists EC' \in ECS$, such that if $EC' = \langle e', C'\rangle$, and $EC = \langle e, C\rangle$, then $e' \equiv e$, and $C' \equiv C$.*

Let $ECS_1, ECS_2 \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, then ECS_1 is subsumed *by ECS_2, denoted with $ECS_1 \sqsubseteq_c ECS_2$, iff each of the sharing sets in ECS_1 is subsumed by ECS_2: $\forall EC \in ECS_1 : EC \leq_c ECS_2$.*

The least upper bound of two collecting sharing sets in $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$ is the union of these sets, modulo equivalence of the underlying components of the domain, here denoted by $\sqcup_c$.

We can show that $\langle \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle), \sqsubseteq_c \rangle$ is a complete lattice where $\bot = \{\,\}$ and $\top = \{\langle e, C\rangle \mid e \in Eqn^+, C \in \wp(\mathcal{SD}_{VI})\}$.

Just like in $\wp(Eqn^+)$ we have the bottom element $\{\,\}$ representing a failing derivation.

6.3.4 Instantiated Concrete Semantics

We now define our concrete Mercury semantics with a domain that specifically keeps track of the sharing information between the data structures that are constructed during the execution of the program.

Definition 6.24 (Concrete Sharing Semantics) *The concrete Mercury semantics with respect to sharing of data structures is defined by the semantic functions $\mathcal{S}em_M$ (Figure 5.4) instantiated with the domain $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$ and the auxiliary functions init_c, comb_c and add_c defined as follows:*

$$
\begin{array}{lcl}
\mathsf{init}_c & = & \{\langle \mathsf{true}, \{\,\}\rangle\} \\
\mathsf{comb}_c(ECS_1, ECS_2) & = & \left\{ \langle e, C_1 \cup C_2\rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \\ \langle e_2, C_2\rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \end{array} \right\} \\
\mathsf{add}_c(unif, ECS) & = & \mathsf{comb}_c(ECS, ECS_{unif})
\end{array}
$$

where `in` *are the input variables of the language construct (the procedure) to which the descriptions ECS_1 and ECS_2 belong, and where $ECS_{unif} = \{\langle E_{unif}, C_{unif}\rangle\}$. E_{unif} is as defined in Definition 5.6 (Page 66), and C_{unif} is given by:*

$$
\begin{array}{lcl}
C_{X:=Y} & = & \{(X^\epsilon - Y^\epsilon)\} \\
C_{X \Leftarrow f(Y_1,\ldots,Y_n)} & = & \{\left(X^{(f,1)} - Y_1^\epsilon\right), \left(X^{(f,2)} - Y_2^\epsilon\right), \ldots, \left(X^{(f,n)} - Y_n^\epsilon\right)\} \\
C_{X \Rightarrow f(Y_1,\ldots,Y_n)} & = & \{\left(X^{(f,1)} - Y_1^\epsilon\right), \left(X^{(f,2)} - Y_2^\epsilon\right), \ldots, \left(X^{(f,n)} - Y_n^\epsilon\right)\} \\
C_{X==Y} & = & \{\,\}
\end{array}
$$

The restriction $C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}}$ in the definition of comb_c is needed to express the fact that, given concrete sharing information for a procedure, this procedure can not add new sharing nor change the sharing between entities that where already instantiated when the procedure was called. Indeed, if two variables X and Y are already bound to some terms (thus fully ground) before the call to a procedure, and say that there is no sharing between the data structures X^{s_X} and Y^{s_Y} before that call, then no matter what that procedure does with these data structures, it can not alter the fact that the memory that each of these data structures represents is unshared.

Example 6.15 *Consider the procedure definition:*

```
%:- pred nothing(T,T,T).
%:- mode nothing(in,in,in) is det.
nothing(X,Y,Z) :- true.
```

where true *is a procedure call that always succeeds. Consider the natural semantics for Mercury (Figure 5.4). Applying* $\mathbf{Pr}_M$ *to the above procedure with call atom* nothing(A,B,C) *and call description*

$$\mathcal{S} = \{\langle e, \{(A^{s_A} - B^{s_B})\}\rangle, \langle e, \{(B^{s_B} - C^{s_C})\}\rangle\}$$

where $e = (A = \tau \wedge B = \tau \wedge C = \tau)$ *and* $\tau \in \mathcal{T}(\mathcal{V}, \Sigma)$*, then*

$$\begin{array}{lcll} \mathcal{S}_0 & = & \{\langle e_1, \{(X^{s_A} - Y^{s_B})\}\rangle, \langle e_1, \{(Y^{s_B} - Z^{s_C})\}\rangle\} & \text{– Renaming} \\ \mathcal{S}_1 & = & \mathcal{S}_0 & \text{– Nothing happens in the goal.} \end{array}$$

with $e_1 = (X = \tau \wedge Y = \tau \wedge Z = \tau)$*. And thus* $\mathsf{comb}_c(\mathcal{S}, \rho_{h\to a}((\mathcal{S}_1)|_h)) = \mathsf{comb}_c(\mathcal{S}, \mathcal{S})$*. Without the special restriction in the definition of* comb_c*, we have:*

$$\mathsf{comb}_c(\mathcal{S}, \mathcal{S}) = \left\{ \begin{array}{l} \langle e, \{(A^{s_A} - B^{s_B})\}\rangle, \\ \langle e, \{(B^{s_B} - C^{s_C})\}\rangle, \\ \underline{\langle e, \{(A^{s_A} - B^{s_B}), (B^{s_B} - C^{s_C})\}\rangle} \end{array} \right\}$$

The underlined new concrete sharing set means that a call to procedure nothing/3*—a procedure that really does nothing—can create additional sharing. Indeed:*

$$(A^{s_A} - C^{s_C}) \preceq_c \{(A^{s_A} - B^{s_B}), (B^{s_B} - C^{s_C})$$

This is not the behaviour we want, especially as all variables of the call were already bound!

Using the definition for comb_c *given in Definition 6.24, we have:*

$$\mathtt{in}(\mathtt{nothing}(\mathtt{A},\mathtt{B},\mathtt{C})) = \{A, B, C\}$$

and thus

$$\begin{array}{lcl} (\{(A^{s_A} - B^{s_B})\})|_{\{A,B,C\}} & = & \{(A^{s_A} - B^{s_B})\} \\ (\{(B^{s_B} - C^{s_C})\})|_{\{A,B,C\}} & = & \{(B^{s_B} - C^{s_C})\} \end{array}$$

As $\{(A^{s_A} - B^{s_B})\} \not\subseteq_c \{(B^{s_B} - C^{s_C})\}$ *we obtain:* $\mathsf{comb}_c(\mathcal{S}, \mathcal{S}) = \mathcal{S}$*. As desired, no additional sharing relations are created between the input arguments of the called procedure.*

We now prove the important theorem stating the equivalence between the natural semantics and the differential semantics for the particular case of this concrete structure sharing domain. This proof is essential, as it allows us to define the abstract structure sharing semantics in the differential context only and prove that

this semantics is correct w.r.t. to concrete differential semantics, which automatically proves that it is correct w.r.t. the natural concrete structure sharing semantics. This is essential, as not only will that enable us to define structure sharing analysis in the context of modules (given the equivalence of the differential semantics with the goal-independent based semantics), but we will show that abstract structure sharing in the context of the natural semantics will in general result in less precise descriptions than in the differential semantics context.

Theorem 6.1 *The natural concrete sharing semantics for Mercury is equivalent with the differential semantics instantiated with the same concrete domain,* i.e.*,*

$$\mathcal{S}em_M(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)) \overset{c}{\Leftrightarrow} \mathcal{S}em_{M\delta}(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle))$$

Proof We must show that each of the statements in Theorem 5.4 holds for our concrete domain. We use $EC, EC_1, \ldots$ to represent elements in $\wp(\mathcal{SD}_{VI})$ and $ECS, ECS_1, \ldots$ for elements in $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$. In this proof we will often refer to Theorem 5.5 (page 88) that states that

$$\mathcal{S}em_M(\wp(Eqn^+)) \overset{c}{\Leftrightarrow} \mathcal{S}em_{M\delta}(\wp(Eqn^+))$$

1. Neutral element (Condition 5.5):

$$\begin{array}{rcl} \mathsf{comb}_c(ECS, \mathsf{init}_c) & = & \mathsf{comb}_c(ECS, \{\langle \mathsf{true}, \{\,\}\rangle\}) \\ & = & \{\langle e \wedge \mathsf{true}, C \cup \{\,\} \mid \langle e, C\rangle \in ECS\} \\ & = & ECS \end{array}$$

2. Associativity (Condition 5.6): This follows immediately from the associativity of the underlying operations, namely the combination of ex-equations using the boolean conjunction operation, and the union operation used for merging the sharing sets.
3. Additivity (Condition 5.7):

$$\begin{array}{rcl} & & \mathsf{comb}_c(ECS_1, ECS_2 \sqcup_c ECS_3)\} \\ & = & \left\{ \langle e, C_1 \cup C_2\rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \\ \langle e_2, C_2\rangle \in ECS_2 \text{ or } \langle e_2, C_2\rangle \in ECS_3, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \end{array} \right\} \\ & = & \left\{ \langle e, C_1 \cup C_2\rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \\ \left(\begin{array}{l} \langle e_2, C_2\rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \end{array} \right) \\ \text{or} \\ \left(\begin{array}{l} \langle e_2, C_2\rangle \in ECS_3, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \end{array} \right) \end{array} \right\} \\ & = & \mathsf{comb}_c(ECS_1, ECS_2) \sqcup_c \mathsf{comb}_c(ECS_1, ECS_3) \end{array}$$

4. Head variable idempotence (Condition 5.8): The projection of ECS onto the head variables $\mathcal{H}$ must satisfy the modes for p. This means that only input variables can be further instantiated, hence structure sharing can only occur between input variables:

$$Vars(\,(ECS)|_{\mathcal{H}}) \subseteq \mathtt{in}(p(X_1,\ldots,X_n))$$

This leads to the following derivation:

$$\begin{array}{ll}
& \mathsf{comb}_c(ECS,\,(ECS)|_{\mathcal{H}}) \\
= & \left\{ \langle e, C_1 \cup C_2 \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS, \\ \langle e_2, C_2\rangle \in\, (ECS)|_{\mathcal{H}} \\ \text{and } e = e_1 \wedge e_2, e \text{ is solvable,} \\ C_2|_{\mathtt{in}} \subseteq_c\, C_1|_{\mathtt{in}} \end{array} \right\} \\
\Downarrow & EC' \in\, (ECS)|_{\mathcal{H}} \wedge \mathtt{in} \subseteq \mathcal{H} \Rightarrow (EC')|_{\mathtt{in}} = EC' \\
= & \left\{ \langle e, C_1 \cup C_2 \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS, \langle e_2, C_2\rangle \in\, (ECS)|_{\mathcal{H}} \\ \text{and } e = e_1 \wedge e_2, e \text{ is solvable,} \\ C_2 \subseteq_c\, C_1|_{\mathtt{in}} \end{array} \right\} \\
\Downarrow & C_1 \subseteq_c\, (C_2)|_V \Rightarrow C_1 \subseteq_c C_2, \quad \forall C_1, C_2, V \\
= & \left\{ \langle e, C_1 \cup C_2 \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS, \langle e_2, C_2\rangle \in\, (ECS)|_{\mathcal{H}} \\ \text{and } e = e_1 \wedge e_2, e \text{ is solvable,} \\ C_2 \subseteq_c C_1 \end{array} \right\} \\
\Downarrow & C_1 \subseteq_c C_2 \Rightarrow C_1 \cup C_2 = C_2, \quad \forall C_1, C_2 \\
= & \{EC \mid EC \in ECS\} = ECS
\end{array}$$

5. add-compatibility (Condition 5.9): Let ECS be the result of combining ECS_1 and ECS_2: $ECS = \mathsf{comb}_c(ECS_1, ECS_2)$. We develop the expression $\mathsf{add}_c(\mathrm{unif}, ECS)$ to $\mathsf{comb}_c(\mathsf{comb}_c(ECS_1, ECS_2), ECS_{\mathrm{unif}})$. By the associativity of comb_c, the latter is equivalent to

$$\mathsf{comb}_c(ECS_1, \mathsf{comb}_c(ECS_2, ECS_{\mathrm{unif}}))$$

which, by applying the definition of add_c is equivalent to

$$\mathsf{comb}_c(ECS_1, \mathsf{add}_c(\mathrm{unif}, ECS_2))$$

6. Projection preservation (Condition 5.10): Let $V = Vars(ECS_1)$, then

we have:

$$
\begin{aligned}
&(\mathsf{comb}_c(ECS_1, ECS_2))|_V \\
&= \left\{ EC \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \langle e_2, C_2\rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}}, \\ EC = \langle e, C_1 \cup C_2\rangle \end{array} \right\}\Big|_V \\
&= \left\{ EC \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \langle e_2, C_2\rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}}, \\ EC = \langle e|_V, C_1|_V \cup C_2|_V\rangle \end{array} \right\} \\
&= \left\{ EC \;\middle|\; \begin{array}{l} \langle e_1, C_1\rangle \in ECS_1, \langle e_2, C_2\rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C1|_{\mathtt{in}}, \\ EC = \langle e_1 \wedge e_2|_V, C_1 \cup C_2|_V \end{array} \right\}
\end{aligned}
$$

We have $\mathtt{in} \subseteq V$, therefore

$$C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \Rightarrow (C_2|_V)|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}}$$

which proves that

$$(\mathsf{comb}_c(ECS_1, ECS_2))|_V = \mathsf{comb}_c(ECS_1, ECS_2|_V)$$

Each of the statements holds, therefore we can conclude that the natural concrete sharing semantics is equivalent with the differential concrete sharing semantics.

□

We illustrate the equivalence through the example of annotating the deterministic procedure of append for a particular call description.

Example 6.16 *Consider the definition of the deterministic version of concatenating two lists as defined in Example 4.4. We will study the annotation of that procedure and its program points for the particular call* append(A,B,C) *where* A *is bound to a list [O1,O2], and* B *is bound to a list [O3]. The variables* 01, 02, 03 *refer to specific terms of a type that is irrelevant for this exposition. We assume that there is no structure sharing between the elements of these particular lists.*

In the context of the natural semantics we obtain the following rulebase meaning of the append*-procedure for the particular call descriptions that the initial call description,*

i.e., $\langle A = [O_1, O_2] \wedge B = [O_3], \{\,\}\rangle$, *generates*[3]*:*

	e	C
$\rightarrow$ `append(A,B,C)`		
call:	$A = [O_1, O_2] \wedge B = [O_3]$	$\{\,\}$
exit:	$A = [O_1, O_2] \wedge B = [O_3]$ $\wedge C = [O_1, O_2, O_3]$	$\left\{ \begin{array}{c} \left(C^{([],1)} - A^{([],1)}\right), \\ \left(C^{([],2)\cdot([],1)} - A^{([],2)\cdot([],1)}\right), \\ \left(C^{([],2)\cdot([],2)} - B^{\epsilon}\right) \end{array} \right\}$
$\rightarrow$ `append(Xs,Y,Zs)`		
call:	$X = [O_1, O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \vert X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
exit:	$X = [O_1, O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \vert X_s] \wedge Z_s = [O_2 \vert Y]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right), \\ \left(Z_s^{([],1)} - X_s^{([],1)}\right), \\ \left(Z_s^{([],2)\cdot([],2)} - Y^{\epsilon}\right) \end{array} \right\}$
$\rightarrow$ `append(Xs,Y,Zs)`		
call:	$X = [O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \vert X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
exit:	$X = [O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \vert X_s] \wedge Z_s = Y$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right), \\ \left(Z_s^{\epsilon} - Y^{\epsilon}\right) \end{array} \right\}$

The same call descriptions are encountered when using the differential setting, yet the exit descriptions will now only list the effect of the literals within the procedure definition,

[3]Note that given the deterministic context, each of the collecting sets consists of only one element. We can therefore omit the surrounding parentheses.

without repeating the call description with which it was called:

	e	C
$\rightarrow$ `append(A,B,C)`		
call:	$A = [O_1, O_2] \wedge B = [O_3]$	$\{\,\}$
exit:	$C = [O_1, O_2, O_3]$	$\left\{ \begin{array}{c} \left(C^{([],1)} - A^{([],1)}\right), \\ \left(C^{([],2)\cdot([],1)} - A^{([],2)\cdot([],1)}\right), \\ \left(C^{([],2)\cdot([],2)} - B^{\epsilon}\right) \end{array} \right\}$
$\rightarrow$ `append(Xs,Y,Zs)`		
call:	$X = [O_1, O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
exit:	$Z_s = [O_2 \mid Y]$	$\left\{ \begin{array}{c} \left(Z_s^{([],1)} - X_s^{([],1)}\right), \\ \left(Z_s^{([],2)\cdot([],2)} - Y^{\epsilon}\right) \end{array} \right\}$
$\rightarrow$ `append(Xs,Y,Zs)`		
call:	$X = [O_2] \wedge Y = [O_3]$ $\wedge X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
exit:	$Z_s = Y$	$\{\ (Z_s^{\epsilon} - Y^{\epsilon})\ \}$

The program point annotations obtained for all these call descriptions are listed in Figures 6.6-6.8 for the natural semantics, and in Figures 6.9-6.11 for the differential semantics context.

	$(X = [O_1, O_2] \wedge Y = [O_3]) = e_1$	$\{\,\} = C_1$
1	e_1	C_1
2	$-$	$-$
3	e_1	C_1
4	$e_1 \wedge X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
5	$e_1 \wedge X = [X_e \mid X_s] \wedge Z_s = [O_2 \mid Y]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right), \\ \left(X_s^{([],1)} - Z_s^{([],1)}\right), \\ \left(Y^{\epsilon} - Z_s^{([],2)}\right) \end{array} \right\}$

Figure 6.6: Annotation table for the deterministic procedure append in the natural semantics context. The first line defines the particular call description.

	$(X = [O_2] \wedge Y = [O_3]) = e_2$	$\{\,\} = C_2$
1	e_2	C_2
2	$-$	$-$
3	e_2	C_2
4	$e_2 \wedge X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^\epsilon\right), \\ \left(X^{([],2)} - X_s^\epsilon\right) \end{array} \right\}$
5	$e_2 \wedge X = [X_e \mid X_s] \wedge Z_s = Y$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^\epsilon\right), \\ \left(X^{([],2)} - X_s^\epsilon\right), \\ \left(Y^\epsilon - Z_s^\epsilon\right) \end{array} \right\}$

Figure 6.7: Annotation table for the deterministic procedure append in the natural semantics context. The first line defines the particular call description.

	$(X = [] \wedge Y = [O_3]) = e_3$	$\{\,\} = C_3$
1	e_3	C_3
2	e_3	C_3
3	e_3	C_3
4	$-$	$-$
5	$-$	$-$

Figure 6.8: Annotation table for the deterministic procedure append in the natural semantics context. The first line defines the particular call description.

6.4 An Abstract Domain for Structure Sharing

The concrete domain proposed in the previous section relies on the notion of data structures to keep track of memory sharing. Given a variable X in an environment described by a constraint e, then the data structure $\langle e, X^s \rangle$ denotes the set of memory cells used to store the subterm X^s, where s is a valid selector for the specific term to which X is bound to in e. In the abstract domain, we make abstraction of the exact environment using a context-free approximation for the sharing data structures and approximating the exact selectors of the concrete data structures by using the equivalence classes over selectors instead. We first define our abstract domain with informal hints w.r.t. the concrete meaning of elements of this domain. The formal concretisation function is given afterwards (Definition 6.28).

Let $\overline{s} \in \mathcal{TG}_{\mathtt{type}(X)}$, then we use $X^{\overline{s}}$ to denote all the data structures that could possibly be selected using the selectors in the equivalence class $[s]$ considering that X is bound to any possible valid term of type $\mathtt{type}(X)$.

Example 6.17 *Consider a variable X of type* list(ex) *(*list(T) *and* ex *as defined in Ex-*

	$X = [O_1, O_2] \wedge Y = [O_3]$	$\{\,\}$
1	$\{\,\}$	$\{\,\}$
2	$-$	$-$
3	$\{\,\}$	$\{\,\}$
4	$X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
5	$X = [X_e \mid X_s] \wedge Z_s = [O_2 \mid Y]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right), \\ \left(X_s^{([],1)} - Z_s^{([],1)}\right), \\ \left(Y^{\epsilon} - Z_s^{([],2)}\right) \end{array} \right\}$

Figure 6.9: Annotation table for the deterministic procedure append in the differential semantics context. The first line defines the particular call description.

	$X = [O_2] \wedge Y = [O_3]$	$\{\,\}$
1	$\{\,\}$	$\{\,\}$
2	$-$	$-$
3	$\{\,\}$	$\{\,\}$
4	$X = [X_e \mid X_s]$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right) \end{array} \right\}$
5	$X = [X_e \mid X_s] \wedge Z_s = Y$	$\left\{ \begin{array}{c} \left(X^{([],1)} - X_e^{\epsilon}\right), \\ \left(X^{([],2)} - X_s^{\epsilon}\right), \\ \left(Y^{\epsilon} - Z_s^{\epsilon}\right) \end{array} \right\}$

Figure 6.10: Annotation table for the deterministic procedure append in the differential semantics context. The first line defines the particular call description.

ample 6.1), then $X^{\overline{\epsilon}}$ *represents the set of context-free data structures*

$$\{X^{\epsilon}, X^{([],2)}, X^{([],2)\cdot([],2)}, \ldots,\}$$

i.e., *all the sublists and their elements, including the list itself. The elements of the list without the list nodes,* i.e., $X^{([],1)}$, $X^{([],2)\cdot([],1)}$ *etc., are selected by* $X^{\overline{([],1)}}$.

Definition 6.25 (Abstract Data Structure) *Let* X *be a variable and* $\overline{s_X}$ *such that* $\overline{s_X} \in \mathcal{TG}_{\mathtt{type}(X)}$*, then* $X^{\overline{s_X}}$ *is called an* abstract data structure *corresponding to the minimal selector* $\overline{s_X}$.

An abstract data structure $X^{\overline{s_X}}$ is meant to represent the set of concrete context-free data structures $\{X^s \mid s \in [s_X]\}$. Note that abstract data structures are by definition context-free.

	$X = [] \wedge Y = [O_3]$	$\{\,\}$
1	$\{\,\}$	$\{\,\}$
2	$\{\,\}$	$\{\,\}$
3	$\{\,\}$	$\{\,\}$
4	$-$	$-$
5	$-$	$-$

Figure 6.11: Annotation table for the deterministic procedure append in the differential semantics context. The first line defines the particular call description.

If X is a variable, then we use $\overline{\mathcal{D}}_X$ to denote the set of abstract data structures over X: $\overline{\mathcal{D}}_X = \{X^{\overline{s}} \mid \overline{s} \in \mathcal{TG}_{\mathtt{type}(X)}\}$. If VI is the set of variables of interest, then we generalise the above notation such that $\overline{\mathcal{D}}_{VI}$ denotes the set of abstract data structures over all the variables in VI: $\overline{\mathcal{D}}_{VI} = \{X^{\overline{s_X}} \mid X^{\overline{s_X}} \in \overline{\mathcal{D}}_X, X \in VI\}$.

In analogy to concrete sharing, we represent abstract sharing between two abstract data structures as a tuple $\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right)$ denoting the fact that the abstract data structure $X^{\overline{s_X}}$ shares with $Y^{\overline{s_Y}}$. This relation is commutative: $\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \Leftrightarrow \left(Y^{\overline{s_Y}} - X^{\overline{s_X}}\right)$. Let $\overline{\mathcal{SD}}_{VI} = \{\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \mid X^{\overline{s_X}}, Y^{\overline{s_Y}} \in \overline{\mathcal{D}}_{VI}\}$, then $\wp(\overline{\mathcal{SD}}_{VI})$ is the abstract domain we use for representing sharing information.

We overload the termshift operation defined for the concrete domain in an obvious way for the abstract domain.

Definition 6.26 (Termshift in the Abstract Domain) *Just as in the concrete domain, the termshift operation expands a given set of abstract sharing data structures to a set that explicitly includes the sharing information of the subterms of the involved data structures. We have* $\mathsf{termshift} : \wp(\overline{\mathcal{SD}}_{VI}) \to \wp(\overline{\mathcal{SD}}_{VI})$ *such that:*

$$\mathsf{termshift}(A) \quad = \quad \left\{ \left(X^{\overline{s_X \bullet s}} - Y^{\overline{s_Y \bullet s}}\right) \;\middle|\; \begin{array}{l} \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \in A \text{ and} \\ \overline{s_X \bullet s} \in \mathcal{TG}_{\mathtt{type}(X)} \end{array} \right\}$$

Note that abstract sharing is also only defined for variables having the same type, therefore, requiring that $\overline{s_Y \bullet s} \in \mathcal{TG}_{\mathtt{type}(Y)}$ in the above definition is not necessary.

As type graphs are finite sets, the termshift operation also results in a finite set of sharing abstract data structures.

Example 6.18 *Let X and Y be of type* $\mathsf{list}(\mathsf{T})$*, then*

$$\mathsf{termshift}(\{\left(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right)\}) = \{\left(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right), \left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right)\}$$

Note that $\left(X^{\overline{([],2)}} - Y^{\overline{([],2)}}\right)$ is equivalent to $\left(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right)$ and therefore does not explicitly appear in the termshifted set.

Using this termshift operation we define the order relation in $\wp(\overline{\mathcal{SD}}_{VI})$.

Definition 6.27 (Ordering in $\wp(\overline{\mathcal{SD}}_{VI})$) *Let $a \in \overline{\mathcal{SD}}_{VI}$, $A \in \wp(\overline{\mathcal{SD}}_{VI})$, then a is* subsumed *by A, denoted by $a \leq_a A$, iff $a \in \mathsf{termshift}(A)$, where $\in$ is the usual set-inclusion operation.*

Let $A_1, A_2 \in \wp(\overline{\mathcal{SD}}_{VI})$, then A_1 is subsumed *by A_2, denoted by $A_1 \sqsubseteq_a A_2$, iff each of the elements of A_1 is subsumed by A_2: $\forall a \in A_1 : a \leq_a A_2$.*

The least upper bound of two sets of abstract sharing data structures is the union of these sets, here denoted by $\sqcup_a$.

Note that $\leq_a$ is defined as the simple set-inclusion w.r.t. the termshifted abstract set, but not its transitive closure. This is an essential aspect of our domain and will become explicit through the definition of the concretisation function we use.

We can easily show that $\langle \wp(\overline{\mathcal{SD}}_{VI}), \sqsubseteq_a \rangle$ is a complete lattice with bottom element $\bot_a = \{\,\}$ and top element $\top_a = \overline{\mathcal{SD}}_{VI}$.

Definition 6.28 (Concretisation Function) *We define the concretisation function $\gamma^{\mathcal{S}} : \wp(\overline{\mathcal{SD}}_{VI}) \to \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$ as follows:*

$$\gamma^{\mathcal{S}} A = \left\{ \langle e, C\rangle \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \preceq_c C \Rightarrow \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a A, \\ \langle e, C\rangle \text{ is valid} \end{array} \right\}$$

There are three interesting aspects to this concretisation function.

1. The concrete environments to which the concretisation maps the abstract information are as such irrelevant. The only purpose of the environment is to validate the concrete sharing information. Note that as the concrete domain is restricted to satisfiable ex-equations, it suffices to require that the structure sharing is valid in the context of the ex-equation without verifying the ex-equation itself.

2. The definition shows that the abstract information only reflects *possible* information: the concrete sharing to which the abstract sharing maps to has to be "allowed" by that abstract sharing set, yet not all the sharing reflected by the abstract sharing set has to be present in each of its resulting concrete sharing sets. This is in contrast of the so called *definite* sharing, c.f. Section 6.6.

3. It makes also clear that A and its transitive closure have a different meaning.

We illustrate some of these aspects in the following examples.

Example 6.19 *Using the type definitions of example 6.1 (page 107), consider the variables E_1, E_2 of type* ex.

Consider the abstract sharing relation $\left(E_1^{\overline{(a,2)}} - E_2^{\overline{(b,1)}}\right)$. *This relationship expresses that* if E_1 *is bound to a term with outermost functor* $a/2$ *and* E_2 *is bound to a term with outermost functor* $b/1$, *then the subterms* $E_1^{(a,2)}$ *and* $E_2^{(b,1)}$ might *share. This is exactly expressed by the result of the concretisation function:*

$$\begin{aligned}&\gamma^{\mathcal{S}}\left(\left\{\left(E_1^{\overline{(a,2)}} - E_2^{\overline{(b,1)}}\right)\right\}\right)\\ &\quad = \quad \{\langle e, \{\,\}\rangle \mid e \in Eqn^+\} \sqcup_c \left\{ \langle e, \left\{\left(E_1^{(a,2)} - E_2^{(b,1)}\right)\right\}\rangle \;\middle|\; e \in Eqn^+\right\}\end{aligned}$$

The concretisation includes the empty sharing set, meaning that the abstract sharing also covers the situation where there is no sharing in the concrete domain. This illustrates the fact that our domain is intended to collect possible *sharing among data structures, and not* definite *sharing.*

Example 6.20 *Let* L_1 *and* L_2 *be two variables of type* $\mathsf{list}(\mathsf{ex})$ *(see Example 6.1), then the sharing relationship* $\left(L_1^{\overline{\epsilon}} - L_2^{\overline{\epsilon}}\right)$ *expresses that the lists, their elements, or any parts of their sublists,* may share *in memory. Indeed, let* $e \in Eqn^+$, *then the concretisation contains the concrete sharing sets* $\langle e, \{\,\}\rangle$ — i.e., *no sharing between the lists, the concrete sharing sets*

$$\left\{\begin{array}{l}\langle e, \{(L_1^{\epsilon} - L_2^{\epsilon})\}\rangle,\\ \langle e, \{\left(L_1^{([],2)} - L_2^{\epsilon}\right)\}\rangle,\\ \langle e, \{\left(L_1^{([],2)} - L_2^{([],2)}\right)\}\rangle,\\ \langle e, \{\left(L_1^{([],2)} - L_2^{([],2)\cdot([],2)}\right)\}\rangle,\\ \ldots\\ \langle e, \{\left(L_1^{([],2)\cdot([],2)} - L_2^{\epsilon}\right)\}\rangle,\\ \ldots\end{array}\right\}$$

which express the possible sharing between sublists of L_1 *and* L_2. *It also comprises sets such as*

$$\begin{array}{l}\langle e, \{\left(L_1^{([],1)} - L_2^{([],1)}\right)\}\rangle,\\ \langle e, \{\left(L_1^{([],1)} - L_2^{([],2)\cdot([],1)}\right)\}\rangle,\\ \langle e, \{\left(L_1^{([],1)} - L_2^{([],1)}\right), \left(L_1^{([],2)\cdot([],1)} - L_2^{([],2)\cdot([],1)}\right), \ldots\}\rangle\\ \ldots\end{array}$$

expressing the fact that only some of the elements of L_1 *and* L_2 *are shared in memory. Combinations are also possible: e.g. the concrete sharing set*

$$\langle e, \{\left(L_1^{([],1)} - L_2^{([],1)}\right), \left(L_1^{([],2)} - L_2^{([],2)}\right)\}\rangle$$

— the first element of L_1 is shared with the first element of L_2, and the tails of L_1 and L_2 are also shared in memory — is also an element of the concretisation of the above abstract sharing set.

Note that if two lists share, then all their sublists share. Yet it is impossible to have sharing between the list and its sublist without allowing infinite terms. Therefore, a concrete sharing set like $\langle e, \left\{ \left(L_1^{\epsilon} - L_2^{\epsilon}\right), \left(L_1^{([],2)} - L_2^{\epsilon}\right) \right\} \rangle$, which by transitivity implies the sharing pair $\left(L_1^{\epsilon} - L_1^{([],2)}\right)$, is not possible as $\nexists e \in Eqn^+$ such that the above sharing can occur.

As a final illustration of the concretisation function, and therefore the meaning of abstract sharing relationships, we sketch the sharing between the elements of lists only:

Example 6.21 *Let L_1 and L_2 be again variables of type* list(ex) *(see Example 6.1), then the sharing relation $\left(L_1^{\overline{([],1}} - L_2^{\overline{([],1)}}\right)$ expresses the fact that there might be some sharing between the elements of the two lists (but not between the lists themselves!). The concretisation of this sharing relation is the set of all combinations of concrete sharing between elements of the two lists, except those combinations that by transitive closure imply internal sharing within one of the lists. E.g. the concrete (context-free) sharing set $\left\{ \left(L_1^{([],1)} - L_2^{([],1)}\right), \left(L_1^{([],2)\cdot([],1)} - L_2^{([],1)}\right) \right\}$ is not included in the concretisation of the given abstract sharing set, as this implies the sharing pair $\left(L_1^{([],1)} - L_1^{([],2)\cdot([],1)}\right)$ which is not covered by the given abstract sharing relation.*

If the abstract sharing description explicitly includes $\left(L_1^{\overline{s_{L_1}}}([],1) - L_1^{\overline{s_{L_1}}}([],1)\right)$, then, of course, the concretisation function will include concrete situations where some of the elements of the list are shared in memory.

In order to apply theorem 5.1 about safe approximating semantics we need to guarantee that $(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle), \gamma^{\mathcal{S}}, \wp(\overline{\mathcal{SD}}_{VI}))$ is an insertion, *i.e.*, $\gamma^{\mathcal{S}}$ should be monotonic and co-strict. We prove this in the following lemma.

Lemma 6.1 $(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle), \gamma^{\mathcal{S}}, \wp(\overline{\mathcal{SD}}_{VI}))$ *is an insertion.*

Proof $\langle \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle), \sqsubseteq_c \rangle$ and $\langle \wp(\overline{\mathcal{SD}}_{VI}), \sqsubseteq_a \rangle$ are complete lattices. Therefore we only need to prove that $\gamma^{\mathcal{S}}$ is monotonic and co-strict.

Consider two abstract sets of data structures A_1 and A_2, and two concrete collecting sharing sets ECS_1 and ECS_2. Let $A_1 \sqsubseteq_a A_2$ and $ECS_1 = \gamma^{\mathcal{S}}(A_1)$, and $ECS_2 = \gamma^{\mathcal{S}}(A_2)$. Let $\langle e, C\rangle \in ECS_1$, then by the definition of the concretisation function, $\forall((X^{s_X} - Y^{s_Y}) \preceq_c C) \Rightarrow \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a A_1$. As $A_1 \sqsubseteq_a A_2$ we have: $\forall((X^{s_X} - Y^{s_Y}) \preceq_c C) \Rightarrow \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a A_2$. Thus $EC \in ECS_2$ too. In general: $\forall EC \in ECS_1 \Rightarrow EC \in ECS_2$, therefore $ECS_1 \sqsubseteq_c ECS_2$.

It is easy to show that $\gamma^{\mathcal{S}}$ maps $\overline{\mathcal{SD}}_{VI}$ to $\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$, hence is costrict. Indeed, the concretisation function maps the set of all possible abstract sharing pairs to all possible, yet valid, structure sharing sets.

□

6.4.1 Additional Operations

Before instantiating the auxiliary functions with this abstract domain, we first define projection and renaming.

Definition 6.29 (Projection) *The projection operation for the abstract domain $\wp(\overline{\mathcal{SD}}_{VI})$ is straightforward:*

$$(A)|_V = \left\{ \left(X^{\overline{s_X}} - Y^{\overline{s_Y}} \right) \mid \left(X^{\overline{s_X}} - Y^{\overline{s_Y}} \right) \in A \text{ and } X, Y \in V \right\}$$

where $V \subseteq VI$.

Definition 6.30 (Renaming) *The renaming of a set of abstract sharing data structures is defined in the obvious way:*

$$\rho_{\overline{X}\to\overline{Y}}(A) = \left\{ \left(Y_i^{\overline{s_i}} - Y_j^{\overline{s_j}} \right) \;\middle|\; \begin{array}{l} \left(X_i^{\overline{s_i}} - X_j^{\overline{s_j}} \right) \in A \\ \text{and } (X_i, X_j), (Y_i, Y_j) \in \overline{X} \to \overline{Y} \end{array} \right\}$$

Where $\overline{X}$ and $\overline{Y}$ are two sequences of variables in VI.

6.4.2 Instantiated Auxiliary Functions

If we plan to use $\langle \wp(\overline{\mathcal{SD}}_{VI}), \sqsubseteq_a \rangle$ as a description domain in the context of one of our semantics of Mercury programs, we need to define the auxiliary functions $\mathsf{init}_a, \mathsf{comb}_a$ and add_a, the instantiated auxiliary functions of $\mathsf{init}, \mathsf{comb}$ and add resp.

As we will see, the operations init_a and add_a are pretty straightforward unlike comb_a. We shall first discuss comb_a.

The domain $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$ is closed under transitive closure. We expressed this through the definition of the order relation $\sqsubseteq_c$ defined on elements from $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$. In the abstract domain, we have defined the order relation not w.r.t. the transitive closure operation as this would lead us to a serious loss of expressivity. Consider the following example:

Example 6.22 *Let E_1 and E_2 be two variables pointing to terms of a certain type T. Let L_1, L_2, L_3 be three lists of type $\mathsf{list}(\mathsf{T})$ such that $L_1 = [E_1, E_2], L_2 = [E_1], L_3 = [E_2]$, thus where L_1 and L_2 share the memory occupied by E_1, and L_1 and L_3 share E_2. Clearly*

there is no sharing between L_2 and L_3. Making abstraction of the variable bindings, then in the concrete domain this particular case of sharing would be correctly described by

$$\left\{\left(L_1^{([],1)} - L_2^{([],1)}\right), \left(L_1^{([],2)\cdot([],1)} - L_3^{([],1)}\right)\right\}$$

The only way to correctly describe this situation in the abstract domain is to include the abstract sharing data structures

$$\left\{\left(L_1^{\overline{([],1)}} - L_2^{\overline{([],1)}}\right), \left(L_1^{\overline{([],2)\cdot([],1)}} - L_3^{\overline{([],1)}}\right)\right\}$$

As $\overline{([],2)\cdot([],1)} = \overline{([],1)}$, the above set is equivalent to

$$\left\{\left(L_1^{\overline{([],1)}} - L_2^{\overline{([],1)}}\right), \left(L_1^{\overline{([],1)}} - L_3^{\overline{([],1)}}\right)\right\}$$

If we had defined $\leq_a$ in terms of the transitive closure of the abstract set, then the concretisation of the abstract set given above would also cover $\left(L_2^{([],1)} - L_3^{([],1)}\right)$, i.e., *the elements of L_3 are shared with elements of L_2. This is not what we intended.*

In general, defining the order relation in $\wp(\overline{\mathcal{SD}}_{VI})$ in terms of the transitive closure would lead to tremendous loss of precision, in particular for recursive types. As described in (Mulkers 1991) we can avoid the use of the transitive closure by using the *alternating closure* for combining old with new sharing sets.

We define the alternating closure operation altclos for any set of unordered tuples.

Definition 6.31 (Alternating Closure) *Consider a set of elements S. Then the alternating closure* altclos : $S \times S \to S \times S \to S \times S$ *is defined as:*

$$\mathsf{altclos}(X,Y) = \left\{ (a_0,a_n) \;\middle|\; \begin{array}{l} (a_0,a_1)\cdot(a_1,a_2)\cdot\ldots\cdot(a_{n-1},a_n) \\ \quad \textit{over } X \textit{ and } Y, n \geq 1, \\ \quad \textit{such that } \forall i, 0 < i < n, (a_{i-1},a_i) \in X \cup Y \\ \qquad \left\{ \begin{array}{l} (a_{i-1},a_i) \in X \Rightarrow (a_i,a_{i+1}) \in Y \\ (a_{i-1},a_i) \in Y \Rightarrow (a_i,a_{i+1}) \in X \end{array} \right. \end{array} \right\}$$

The alternating closure is commutative and associative. Indeed, $\mathsf{altclos}(A,B) = \mathsf{altclos}(B,A)$ and both $\mathsf{altclos}(A,\mathsf{altclos}(B,C))$ and $\mathsf{altclos}(\mathsf{altclos}(A,B),C)$ represent the tuples that are obtained by constructing paths alternating over the three sets, *i.e.*, no path contains two consecutive edges stemming from the same initial set A, B, or C.

We can now define our abstract Mercury semantics w.r.t. memory sharing.

Definition 6.32 (Abstract Sharing Semantics) *The abstract Mercury semantics with respect to sharing of data structures is defined as the differential semantics $\mathcal{S}em_{M\delta}$ (Figure 5.6) instantiated with $\langle \wp(\overline{\mathcal{SD}_{VI}}), \sqsubseteq_a \rangle$ as description domain and the following functions* $\mathsf{init}_a, \mathsf{comb}_a$ *and* add_a *as the auxiliary functions* init, comb *and* add *resp.:*

$$\begin{array}{lcl} \mathsf{init}_a & = & \{\} \\ \mathsf{comb}_a(A_1, A_2) & = & \mathsf{altclos}(\mathsf{termshift}(A_1), \mathsf{termshift}(A_2)) \\ \mathsf{add}_a(\mathit{unif}, A) & = & \mathsf{comb}_a(A, A_{\mathit{unif}}) \end{array}$$

where A_{unif} is defined as:

$$\begin{array}{lcl} A_{X:=Y} & = & \{(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}})\} \\ A_{X \Leftarrow f(Y_1,\ldots,Y_n)} & = & \left\{ \left(X^{\overline{(f,1)}} - {Y_1}^{\overline{\epsilon}}\right), \left(X^{\overline{(f,2)}} - {Y_2}^{\overline{\epsilon}}\right), \ldots, \left(X^{\overline{(f,n)}} - {Y_n}^{\overline{\epsilon}}\right) \right\} \\ A_{X \Rightarrow f(Y_1,\ldots,Y_n)} & = & \left\{ \left(X^{\overline{(f,1)}} - {Y_1}^{\overline{\epsilon}}\right), \left(X^{\overline{(f,2)}} - {Y_2}^{\overline{\epsilon}}\right), \ldots, \left(X^{\overline{(f,n)}} - {Y_n}^{\overline{\epsilon}}\right) \right\} \\ A_{X==Y} & = & \{\} \end{array}$$

We must show that the abstract semantics are a safe approximation of the concrete semantics. Applying Theorem 5.1 it suffices to show that $\mathsf{init}_a \propto \mathsf{init}_c$, $\mathsf{comb}_a \propto \mathsf{comb}_c$, and $\mathsf{add}_a \propto \mathsf{add}_c$ to conclude that

$$\mathcal{S}em_{M\delta}(\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)) \propto \mathcal{S}em_{M\delta}(\wp(\overline{\mathcal{SD}_{VI}}))$$

Lemma 6.2 $\mathsf{init}_a \propto \mathsf{init}_c$.

Proof The proof is straightforward. $\mathsf{init}_a = \{\}$, and $\gamma^{\mathcal{S}}(\{\}) = \{\langle e, \{\}\rangle \mid e \in Eqn^+\}$ which clearly subsumes the value $\{\langle \mathsf{true}, \{\}\rangle\} = \mathsf{init}_c$. □

The ordering in the concrete domain is centred around the transitive closure, while in the abstract domain, all sharing information is propagated through the alternating closure. The essence of the proof of $\mathsf{comb}_a \propto \mathsf{comb}_c$ relies in showing that the transitive closure over elements of the union of two concrete sets of sharing structures can always be formulated as the alternative closure over these two sets. This is shown in the following lemma. The lemma is formulated for context-free sharing sets.

Lemma 6.3 *Let $C_1, C_2 \in \wp(\mathcal{SD}_{VI})$.*
If there exists a finite sequence Seq such that

$$\left\{ \begin{array}{l} Seq = (t_0 - t_1), (t_1 - t_2), \ldots, (t_{n-1} - t_n), \\ 0 \leq i < n : (t_i - t_{i+1}) \preceq_c C_1 \cup C_2 \end{array} \right. \tag{6.1}$$

which corresponds to computing a transitive closure, then there exists a finite sequence Seq' *such that*

$$\left\{\begin{array}{l} Seq' = \left(t'_0 - t'_1\right), \left(t'_1 - t'_2\right), \ldots, \left(t'_{m-1} - t'_m\right), \\ t'_0 = t_0, t'_m = t_n, \\ 0 < i < m : \\ \quad \left(t'_{i-1} - t'_i\right) \preceq_c C_1 \Rightarrow \left(t'_i - t'_{i+1}\right) \preceq_c C_2 \\ \quad \textit{and} \\ \quad \left(t'_{i-1} - t'_i\right) \preceq_c C_2 \Rightarrow \left(t'_i - t'_{i+1}\right) \preceq_c C_1 \end{array}\right. \tag{6.2}$$

which corresponds to an alternating sequence.

Proof The lemma and proof are similar to Lemma 4.1.21 in (Mulkers 1991), but as the domains show some differences we give a new specialised proof here.

If Equation (6.1) does not already satisfy Equation (6.2) then there exists a value k, $0 < k < n$, such that $\left(t_{k-1} - t_k\right), \left(t_k - t_{k+1}\right) \preceq_c C_1$ (or C_2). We can apply one of the following reductions:

1. If $t_{k-1} = t_{k+1} = t$ then both tuples can be removed from the sequence, hence

$$Seq = \left(t_0 - t_1\right), \ldots, \left(t_{k-2} - t\right), \left(t - t_{k+2}\right), \ldots \left(t_{n-1} - t_n\right)$$

2. If $t_{k-1} \neq t_{k+1}$, then the sequence $\left(t_{k-1} - t_k\right), \left(t_k - t_{k+1}\right)$ implies the existence of a tuple of sharing data structures such that $\left(t_{k-1} - t_{k+1}\right) \preceq_c C_1$ (similarly for C_2). This means that the original sequence can be reduced to

$$Seq = \left(t_0 - t_1\right), \ldots, \left(t_{k-1} - t_{k+1}\right), \ldots, \left(t_{n-1} - t_n\right)$$

We can repeat these reductions until the sequence does not contain any neighbouring tuples both only subsumed by either C_1 or C_2. This situation satisfies Equation (6.2).

□

Lemma 6.4 *Let* $ECS_1, ECS_2 \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, *and* $A_1, A_2 \in \wp(\overline{\mathcal{SD}}_{VI})$ *such that*

$$ECS_1 \sqsubseteq_c \gamma^{\mathcal{S}}(A_1), ECS_2 \sqsubseteq_c \gamma^{\mathcal{S}}(A_2)$$

Then for all $EC_1 = \langle e_1, C_1\rangle \in ECS_1$ *and* $EC_2 = \langle e_2, C_2\rangle \in ECS_2$, *if* $e = e_1 \wedge e_2$ *and* e *is solvable, then* $\forall \left(X^{s_X} - Y^{s_Y}\right)$*: if* $\left(X^{s_X} - Y^{s_Y}\right) \preceq_c \left(C_1 \cup C_2\right)$ *then* $\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a \mathsf{comb}_a(A_1, A_2)$.

Proof If $(X^{s_X} - Y^{s_Y}) \preceq_c (C_1 \cup C_2)$ then this means that there exists a sequence $Seq = (t_1{}^{s_{t_1}} - t_2{}^{s_{t_2}}) \ldots, (t_{n-1}{}^{s_{t_{n-1}}} - t_n{}^{s_{t_n}})$ satisfying Equation (6.1) w.r.t. the sharing information in C_1 and C_2, and $t_1{}^{s_{t_1}} = X^{s_X}$ and $t_n{}^{s_{t_n}} = Y^{s_Y}$. By Lemma 6.3 this means that Seq can be transformed into a sequence

$$Seq' = \left(t'_1{}^{s_{t'_1}} - t'_2{}^{s_{t'_2}}\right) \ldots, \left(t'_{m-1}{}^{s_{t'_{m-1}}} - t'_m{}^{s_{t'_m}}\right)$$

alternating between elements subsumed by the sharing information in EC_1 and elements subsumed by the sharing part of EC_2, and $t'_1{}^{s_{t'_1}} = X^{s_X}$ and $t'_m{}^{s_{t'_m}} = Y^{s_Y}$. As $ECS_1 \sqsubseteq_c \gamma^{\mathcal{S}}(A_1)$ and $ECS_2 \sqsubseteq_c \gamma^{\mathcal{S}}(A_2)$, we know that for each $k, 2 \leq k \leq m$, if $\left(t'_{k-1}{}^{s_{t'_{k-1}}} - t'_k{}^{s_{t'_k}}\right) \preceq_c C_1$, then $\left(t'_{k-1}{}^{\overline{s_{t'_{k-1}}}} - t'_k{}^{\overline{s_{t'_k}}}\right) \sqsubseteq_a A_1$, idem for C_2 and A_2 (definition of the concretisation function). This means that there exists an sequence $\left(t'_1{}^{\overline{s_{t'_1}}} - t'_2{}^{\overline{s_{t'_2}}}\right), \ldots, \left(t'_{n-1}{}^{\overline{s_{t'_{n-1}}}} - t'_n{}^{\overline{s_{t'_n}}}\right)$ alternating over elements that are subsumed by A_1 and A_2, *i.e.*, elements of the sets $\mathsf{termshift}(A_1)$, and $\mathsf{termshift}(A_2)$, where $t'_1{}^{\overline{s_{t'_1}}} = X^{\overline{s_X}}$ and $t'_n{}^{\overline{s_{t'_n}}} = Y^{\overline{s_Y}}$. This implies that $\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a \mathsf{altclos}(\mathsf{termshift}(A_1), \mathsf{termshift}(A_2))$, thus we have $\left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a \mathsf{comb}_a(A_1, A_2)$.

□

Using the previous two lemma's we come to the central lemma, being that the abstract combination operation is a safe approximation of the concrete combination operation:

Lemma 6.5 $\mathsf{comb}_a \propto \mathsf{comb}_c$.

Proof Applying the definition of $\propto$ (Definition 5.2), we need to prove that $\forall A_1, A_2 \in \wp(\overline{\mathcal{SD}}_{VI}), \forall ECS_1, ECS_2 \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle) : ECS_1 \sqsubseteq_c \gamma^{\mathcal{S}}(A_1), ECS_2 \sqsubseteq_c \gamma^{\mathcal{S}}(A_2) \Rightarrow \mathsf{comb}_c(ECS_1, ECS_2) \sqsubseteq_c \gamma^{\mathcal{S}}(\mathsf{comb}_a(A_1, A_2))$.

With the definition for $\mathsf{comb}_c(ECS_1, ECS_2)$, we have:

$$
\begin{array}{ll}
& \mathsf{comb}_c(ECS_1, ECS_2) \\
= & \left\{ \langle e, C_1 \cup C_2 \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1 \rangle \in ECS_1, \langle e_2, C_2 \rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ C_2|_{\mathtt{in}} \subseteq_c C_1|_{\mathtt{in}} \end{array} \right\} \\
\sqsubseteq_c & \left\{ \langle e, C_1 \cup C_2 \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1 \rangle \in ECS_1, \langle e_2, C_2 \rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable} \end{array} \right\} \\
\sqsubseteq_c & \left\{ \langle e, C \rangle \;\middle|\; \begin{array}{l} \langle e_1, C_1 \rangle \in ECS_1, \langle e_2, C_2 \rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, C = C_1 \cup C_2 \end{array} \right\} \\
\sqsubseteq_c & \left\{ \langle e, C \rangle \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \preceq_c C \\ \quad \Rightarrow (X^{s_X} - Y^{s_Y}) \preceq_c (C_1 \cup C_2), \\ \langle e_1, C_1 \rangle \in ECS_1, \langle e_2, C_2 \rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable} \end{array} \right\} \\
\Downarrow & \text{Lemma 6.4}, ECS_1 \sqsubseteq_c \gamma^{\mathcal{S}}(A_1), ECS_2 \sqsubseteq_c \gamma^{\mathcal{S}}(A_2) \\
\sqsubseteq_c & \left\{ \langle e, C \rangle \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \preceq_c C \\ \quad \Rightarrow (X^{s_X} - Y^{s_Y}) \preceq_c (C_1 \cup C_2) \\ \quad \Rightarrow \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a \mathsf{comb}_a(A_1, A_2), \\ \langle e_1, C_1 \rangle \in ECS_1, \langle e_2, C_2 \rangle \in ECS_2, \\ e = e_1 \wedge e_2, e \text{ is solvable}, \\ ECS_1 \sqsubseteq_c \gamma^{\mathcal{S}}(A_1), ECS_2 \sqsubseteq_c \gamma^{\mathcal{S}}(A_2) \end{array} \right\} \\
\sqsubseteq_c & \left\{ \langle e, C \rangle \;\middle|\; \begin{array}{l} (X^{s_X} - Y^{s_Y}) \rangle \preceq_c C \\ \quad \Rightarrow \left(X^{\overline{s_X}} - Y^{\overline{s_Y}}\right) \leq_a \mathsf{comb}_a(A_1, A_2) \end{array} \right\} \\
\Downarrow & \text{Definition of the concretisation function.} \\
\sqsubseteq_c & \gamma^{\mathcal{S}}(\mathsf{comb}_a(A_1, A_2))
\end{array}
$$

This proves the lemma.

□

Finally, we need to prove that the abstract operation add_a safely approximates the concrete operation add_c. This is done in the following lemma.

Lemma 6.6 $\mathsf{add}_a \propto \mathsf{add}_c$.

Proof Both add_a and add_c are defined in terms of the combination operator, comb_a resp. comb_c. It suffices to show that for all unifications unif, $ECS_{\text{unif}} \sqsubseteq_c \gamma^{\mathcal{S}}(A_{\text{unif}})$ where ECS_{unif} is as defined in Definition 6.24, and A_{unif} as in Definition 6.32. The proof is trivial. Take for example the unification $X := Y$. In the abstract domain this yields $A_{X:=Y} = \{\left(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right)\}$, the concretisation of which clearly includes the concrete sharing set $\langle X = Y, \{(X^{\epsilon} - Y^{\epsilon})\}\rangle$. The same conclusions can be drawn for the remaining three cases of unification.

□

Lemma 6.2, Lemma 6.5 and Lemma 6.5 allow us to conclude with the theorem:

Theorem 6.2 (Safe Sharing Approximation) *$\mathcal{S}em_{M\delta}(\wp(\overline{\mathcal{SD}}_{VI}))$ is a safe approximation for the concrete sharing semantics $\mathcal{S}em_{M\delta}(\wp(\langle Eqn^{+}, \wp(\mathcal{SD}_{VI})\rangle))$.*

Example 6.23 *In Example 6.16 we studied the deterministic version of* append *(defined in Example 4.4) in the concrete semantics. In this example we repeat the exercise, yet now in the abstract semantics.*

In the concrete domain, the initial call for append(A,B,C) *was $\langle A = [O_1, O_2] \wedge B = [O_3], \{\,\}\rangle$, which in the abstract domain can be safely approximated by $\{\,\}$, reflecting the absence of any structure sharing among the input variables of the call. We obtain the following rulebase meaning for each of the encountered call descriptions:*

	A
$\rightarrow$ `append(A,B,C)`	
call:	$\{\,\}$
exit:	$\{\left(C^{\overline{([],1)}} - A^{\overline{([],1)}}\right), (C^{\overline{\epsilon}} - B^{\overline{\epsilon}})\}$
$\rightarrow$ `append(Xs,Y,Zs)`	
call:	$\{\left(X^{\overline{([],1)}} - X_e{}^{\overline{\epsilon}}\right), (X^{\overline{\epsilon}} - X_s{}^{\overline{\epsilon}})\}$
exit:	$\{\left(Z_s{}^{\overline{([],1)}} - X_s{}^{\overline{([],1)}}\right), (Z_s{}^{\overline{\epsilon}} - Y^{\overline{\epsilon}})\}$

As can be seen, the exit descriptions for both calls are the same (upon renaming). This is exactly what is expected from the differential semantics in the abstract setting.

The program point annotations are depicted in Figure 6.12.

	$\{\,\}$
1	$\{\,\}$
2	$\{\,\}$
3	$\{\,\}$
4	$\{\left(X^{\overline{([],1)}} - X_e{}^{\overline{\epsilon}}\right), (X^{\overline{\epsilon}} - X_s{}^{\overline{\epsilon}})\}$
5	$\{\left(X^{\overline{([],1)}} - X_e{}^{\overline{\epsilon}}\right), (X^{\overline{\epsilon}} - X_s{}^{\overline{\epsilon}}), (Y^{\overline{\epsilon}} - Z_s{}^{\overline{\epsilon}})\}$

Figure 6.12: Annotation table for append in the abstract differential semantics context. The first line defines the particular call description.

Given the general equivalence between the differential semantics and the goal-independent based semantics, we have:

$$\mathcal{S}em_{M\delta}(\wp(\overline{\mathcal{SD}}_{VI})) \Leftrightarrow \mathcal{S}em_{M\star}(\wp(\overline{\mathcal{SD}}_{VI}))$$

6.5 The Analysis System

In our multiple prototypes we have implemented the structure sharing analysis using the above described notions and operations. Given the equivalence of the abstract sharing semantics with the goal-independent based instantiation with that same abstract domain, we have obviously chosen to implement structure sharing using the goal-independent based semantics. This has the advantage that for every procedure only one exit description needs to be computed. This is not the case in the differential semantics settings, as is also illustrated by the rulebase meaning obtained for the deterministic version of `append` described in Example 6.23. Moreover, analysis in the presence of modules becomes straightforward as it suffices to record the goal-independent analysis results of each of the analysed modules into dedicated files (the so called optimisation interface files). These results can then be used for the analysis of procedures that depend on procedures defined in other modules, without having to consult the entire source code of these other modules.

6.6 Related Work

The notion of *sharing* in the context of logic programming languages is often used to refer to *variable sharing* instead of memory structure sharing as we used the term here. Variable sharing is an extensively studied property of logic programs enabling many different transformations and optimisations of these language: it is for example essential for the efficient exploitation of AND-parallelism, it is also important for optimising the efficiency of general unification, it also appears indirectly in the computation of other properties of interest, such as for example freeness information, etc. The most known sharing domain is the domain introduced by (Jacobs and Langen 1992) for which many variations have been studied. See (Bagnara, Zaffanella, and Hill 2005) for a survey of the different sharing analysis techniques that are current now.

Although sharing also collects some form of shared memory (the free variables), the behaviour of the operations defined on the domains representing such sharing is completely different from the behaviour we need for structure sharing. Indeed, during the execution of a logic program, free variables become instantiated. This means that the number of pairs of variables that may possibly share some variables will diminish as execution goes on. This behaviour will also be reflected in the abstract domain describing variable sharing. In our case of structure sharing, we have a somewhat opposite behaviour: as execution goes on, new terms are built or transformed, which means that more and more memory blocks are shared by the same terms. Hence, the size of the set of possibly shared terms will have the general tendency of growing during the evaluation of a predicate instead of shrinking like is more the case with variable sharing. Therefore, in-

tuitively, we think that while the underlying basic representation of the variable sharing information could in some sense form an inspiration for representing memory sharing, the operations and therefore the precision enhancements developed for variable sharing are not of great use in our context.

In the variable sharing literature, one can also find the notion of *structural information* (Bagnara, Hill, and Zaffanella 2000). Here, the structural information is not so much the structure of the terms that are constructed in the program, yet is used to describe the structure of the calls to given predicates. A typical example (Codish, Marriott, and Taboch 2000) is to statically detect the use of difference lists. If a predicate is always called with a difference list as one of its arguments, then the precision of the variable sharing results can be significantly increased if such a predicate is first transformed. Yet variable sharing, even in the presence of structure information, can not be used to derive the run-time property of memory sharing (Bagnara, Hill, and Zaffanella 2000).

6.7 Conclusion

In this chapter we have defined a formal setting for discussing the fact that terms pointed at by different variables may occupy the same heap space at run-time. This notion is important as we need to be able to trace back to all the variables that may point to a particular block of heap space, in order to guarantee that that heap space may or may not be reused for other purposes. We introduced the notions of data structures and sharing pairs, in a concrete setting, as well as in an abstract setting.

Using these domains, we defined the concrete natural structure sharing semantics of Mercury programs as an instantiation of $\mathcal{S}em_M$, and the abstract structure sharing semantics as an instantiation of $\mathcal{S}em_{M\delta}$. We showed that in the concrete domain, the natural semantics is equivalent to the differential semantics, which by transitivity demonstrates the correctness of the abstract differential semantics w.r.t. the concrete natural semantics of Mercury.

Chapter 7

In Use Information

This chapter defines the notions of forward use and backward use variables. This information is needed for deducing liveness information.

7.1 Introduction

If one halts the execution of a program just after executing the literal at some specific program point —let us call this program point the *current* program point—, then the instantiated heap cells[1] at that moment can be put into two categories. Either they are still used by a literal executed after the literal at the current program point, or they are not. If they are not used anymore, then this means that no other instruction during the further execution of the program needs to access the information kept in them. Hence, the literal at the current program point may have had the last and unique access to some of these unused heap cells in which case these heap cells are said to be *dead* after the current literal. On the other hand, if some subsequent instruction does access the cells, then these heap cells are still in use. We call such heap cells *live* heap cells. Liveness can be due to two basic computation mechanisms:

1. the cells are accessed because any of the goals belonging to the same execution path as the current program point, and following the literal at the current program point, is using the cells. This is called *forward use*;
2. Mercury is a logic programming language, which means that in the presence of non-deterministic code, *i.e.*, backtracking, goals defined *prior* to the current program point, may still be needing the values stored in those heap cells. This is called *backward use*.

[1] The heap may of course be larger than the set of heap cells that is actually *filled*. The *empty* heap cells are of no interest here.

If we call the procedure definition to which the current program point belongs the *current procedure definition*, then we can split heap cells in forward or backward use each into two groups, namely: forward and backward use *local* w.r.t. the current procedure, and forward and backward use that is *global* w.r.t. that procedure definition and the particular call with which it was called. The former group consists of the heap cells that are accessed by any of the goals *within* the current procedure definition, while the latter are heap cells accessed by any other goal. Of course, these two groups may overlap.

Example 7.1 *Consider the following fragment of Mercury code.*

```
%:- pred member(t, list(t)).
%:- mode member(out, in) is nondet.
member(E, L):-
        (1) L => [F|R],
        (
                (2) E := F
        ;
                (3) member(E,R)
        ).

%:- pred generate(t, list(t)).
%:- mode generate(out, out) is nondet.
generate(X,LL):-
        (4) LL <= [f(1),f(2),g(3)],
        (5) member(X, LL).
```

We study the state of the relevant heap cells when execution reaches program point (2) *a first time,* i.e.*, when* member/2 *is called with the second argument L bound to the same list as pointed at by LL, and when no backtracking has occurred yet.*

- local forward use: *after successful completion of the assignment of F to E, the set of instantiated variables becomes* $\{L, F, R, E\}$*. As program point* (2) *belongs to the execution path* $(1) - (2)$ *and is therefore not followed by any other literals, none of these instantiated variables is in* local forward use *w.r.t. program point* (2)*.*

- local backward use: member/2 *is non-deterministic, which means that backtracking may occur and that some data structures, although not in (local) forward use, might still be of importance in the presence of backtracking. Concretely, at program point* (2)*, all the structures that the variables from the second branch may point to can be seen as being in backward use, hence the set* $\{E, R\}$*.*

- global forward/backward use: *the global forward/backward use at* (2) *(or at any program point within the same procedure definition) is determined by the local and global forward/backward use at program point* (5)*,* i.e.*, the context in which* member/2 *was called. As* generate/2 *has two output arguments, chances are that*

these arguments will be in forward use w.r.t. the program point where generate *is called. Hence, as an end effect, both E and L can be in global forward use at* (2).

In this chapter we focus on the local use. We will see that the global components are automatically taken into account when propagating liveness information (Chapter 8).

In the subsequent sections we abbreviate local forward use and local backward use to forward use and backward use respectively.

7.2 Forward Use

Given a program point (i), called the *current program point*, and the procedure definition to which (i) belongs, called the *current procedure definition*, we say that a variable, say X, is in forward use w.r.t. (i) if:

- X is an instantiated variable after performing the literal at (i). These are the variables that either get instantiated by one of the literals preceding the current literal or the literal itself, or are variables that are input to the current procedure;
- and X is used in one of the literals following the literal at the current program point and covered by the same execution path as to which the current program point belongs.

In a more formal way, the variables in forward use at a program point (i) in a procedure $h \leftarrow g$, denoted by $\mathsf{forward} : \mathtt{pp} \rightarrow \wp(VI)$, where VI are the variables of interest in the considered program, are defined as:

$$\mathsf{forward}(i) = \begin{array}{c} \left(\left(\bigcup_{j \in \mathtt{pre}(i)} \mathtt{out}(l_j)\right) \cup \mathtt{out}(l_i) \cup \mathtt{in}(h \leftarrow g)\right) \\ \bigcap \\ \left(\bigcup_{j \in \mathtt{post}(i)} \mathit{Vars}(l_j)\right) \end{array}$$

The expression $\cup_{j \in \mathtt{pre}(i)} \mathtt{out}(l_j)$ represents the variables that get instantiated by one of the literals preceding program point (i), the set of variables instantiated at (i) is the set $\mathtt{out}(l_i)$, and the set $\mathtt{in}(h \leftarrow g)$ returns the input variables of the procedure to which (i) belongs. The forward use variables are the variables from the set of instantiated variables given by the union of the variables in the previous sets and those variables that also occur in one of the literals following (i).

We generalise the above definition to goals instead of individual program points: Let g' be a subgoal of the goal in the procedure definition $h \leftarrow g$, then:

$$\mathsf{forward}(g') = \begin{array}{c} \left(\left(\bigcup_{j \in \mathtt{pre}(g')} \mathtt{out}(l_j)\right) \cup \mathtt{out}(g') \cup \mathtt{in}(h \leftarrow g)\right) \\ \bigcap \\ \left(\bigcup_{j \in \mathtt{post}(g')} \mathit{Vars}(l_j)\right) \end{array}$$

Note that forward use is a syntactic property of procedure definitions. The notions of concrete or abstract domain are irrelevant here.

Example 7.2 *Considering the program fragment given in Example 7.1, the following table lists for each program point the set of variables that are instantiated after completion of the literal corresponding to that program point (I_i), the set of variables that are used by any of the literals following a program point (U_i), and the set of variables in forward use for that program point (F_i).*

pp	I_i	U_i	$F_i = I_i \cap U_i$
1	$\{L, F, R\}$	$\{E, F, R\}$	$\{F, R\}$
2	$\{L, F, R, E\}$	$\{\}$	$\{\}$
3	$\{L, F, R, E\}$	$\{\}$	$\{\}$
4	$\{LL\}$	$\{X, LL\}$	$\{LL\}$
5	$\{X, LL\}$	$\{\}$	$\{\}$

Example 7.3 *For the deterministic version of* append *of Example 4.4, we obtain the forward use information tabled below:*

pp	I_i	U_i	$F_i = I_i \cap U_i$
1	$\{X, Y\}$	$\{Z, Y\}$	$\{Y\}$
2	$\{X, Y, Z\}$	$\{\}$	$\{\}$
3	$\{X, Y, Xe, Xs\}$	$\{Xs, Y, Zs, Xe, Z\}$	$\{Y, Xe, Xs\}$
4	$\{X, Y, Xe, Xs, Zs\}$	$\{Z, Xe, Zs\}$	$\{Xe, Zs\}$
5	$\{X, Y, Z, Xe, Xs, Zs\}$	$\{\}$	$\{\}$

with I_i, U_i, F_i have the same meaning as in the previous example.

7.3 Backward Use

A variable X is said to be in backward use w.r.t. a program point (i) within a procedure definition p if that variable is instantiated at (i) and can be accessed after backtracking has reentered the code prior to (i). By *accessing* we mean that the value to which the variable is bound at (i) is input to some literal that may be executed after (i). When we restrict this access to literals belonging to the definition of procedure p, then we obtain the variables in local backward use.

In this definition we use variables as the descriptions for the memory cells that they refer to. A more fine grained approach to backward use information consists of using data structures instead, hence parts of the memory cells that the variables may point to. This may indeed add some form of precision, yet comes at the cost of a full program analysis to correctly derive backward use information. We briefly illustrate and sketch this approach in Section 7.4.

The derivation of backward use information is more complex than for forward use. We use the same denotational approach as we did earlier, yet given the fact that backward use information is due to non-deterministic goals instead of the built-in literals (unifications), we need to define a separate semantics for Mercury programs.

7.3.1 Basic Denotational Definition

The goal is to record backward use information for each program point in the program. As we collect variables, we construct an annotation table with signature:

$$Ann_b = \mathsf{pp} \rightarrow \wp(VI)$$

where VI is the set of variables of interest in our programs.

We first describe backward use for procedures, goals and literals.

Procedures. The signature of the semantic function $\mathbf{Pr}_b$ for procedures is given by:

$$\mathbf{Pr}_b : \mathit{Procedure} \rightarrow \mathit{Ann}_b \rightarrow \mathit{Ann}_b$$

Let B be the annotation table mapping individual program points to their backward use annotation, then we define backward use for such a procedure as:

$$\mathbf{Pr}_b[\![h \leftarrow g]\!]B \quad = \quad \begin{array}{l}\mathsf{let}\ (b_1, B_1) = \mathbf{G}_b[\![g]\!]B\ \mathsf{init}_b\ \mathsf{in} \\ \qquad B_1\end{array}$$

In this domain $\mathsf{init}_b = \{\ \}$, *i.e.*, initially there are no variables considered to be in backward use w.r.t. the main goal of the procedure.

Goals. To derive backward use for a goal, we need the following information:

- The annotation table mapping program points to backward use information. This table needs to be updated. We use the symbol B.
- The set of variables that are already in backward use and are due to the goals that are prior to the current goal – b.

Using this information and of course the goal itself we compute a new set of variables that are in backward use w.r.t. this goal. We also update the annotation table. The signature for $\mathbf{G}_b$ is therefore:

$$\mathbf{G}_b : \mathit{Goal} \rightarrow \mathit{Ann}_b \rightarrow \wp(VI) \rightarrow \wp(VI) \rightarrow \mathit{Ann}_b$$

We define $\mathbf{G}_b$ for backward use information as follows:

- *conjunction.* For a conjunction, we accumulate the collected backward use information:

$$\begin{array}{rcl} \mathbf{G}_b[\![g_1, g_2]\!] B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!] B\ b \text{ in} \\ & & \quad \mathbf{G}_b[\![g_2]\!] B_1\ b_1 \end{array}$$

- *disjunction.* For a disjunction we need to differentiate two cases: deterministic selection and general non-deterministic disjunction. If the disjunction represents a deterministic selection, then once one of the branches is selected, there is no backtracking to any of the other branches possible. This means that a deterministic selection does not add any additional backward use variables by itself, thus:

$$\begin{array}{rcl} \mathbf{G}_b[\![g_1; g_2]\!] B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!] B\ b \text{ in} \\ & & \quad \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!] B_1\ b \text{ in} \\ & & \quad\quad (b_1 \cup b_2, B_2) \\ & & \quad\quad \text{if } \texttt{switch}((g_1; g_2)) = \mathsf{true} \end{array}$$

  After annotating each of the branches, the resulting set of backward use variables of the disjunction is the union of the backward use variables of each of the branches.

  When a disjunction is not a switch then backtracking is possible. In this case, we need to clearly specify the backward use information passed to the first branch of the disjunction, the backward use information passed to the second branch, and finally, the backward use information that is returned as a result of the disjunction and that will be used for computing backward use in subsequent goals:

  - *First branch.* Clearly, all the input variables to the second branch are all variables that will definitely be accessed if that second branch is executed. Also the variables in forward use w.r.t. that second branch will be accessed when the computation using the first branch fails. Therefore, all these variables are added to the already existing set of backward use variables.
  - *Second branch.* Once the second branch is performed, no choice points are left, and therefore, execution has no alternative branch to perform in that disjunction. This means that for the second branch, no extra variables need to be added to the backward use set. Note that this is in accordance with the selection rule being left-to-right.
  - *Result.* As execution may come back to this disjunction, we must guarantee that all variables used after that disjunction remain unaltered. For this purpose we could explicitly add all the variables in forward use w.r.t. the disjunction to the resulting set of variables in backward use. Observe that these variables are already in $b_1 \cup b_2$. Indeed, due to

the strongly moded character of Mercury all variables used beyond the scope of a disjunction must have the same instantiation after each of the branches of that disjunction, hence $\mathsf{forward}(g_1; g_2) = \mathsf{forward}(g_1) = \mathsf{forward}(g_2)$. As $\mathsf{forward}(g_2)$ is explicitly added to the set I which becomes a subset of b_1 we have: $\mathsf{forward}(g_1; g_2) \subseteq b_1 \cup b_2$. Therefore, nothing needs to be added to the union of both backward use sets.

We obtain the following semantic rule for non-deterministic disjunctions:

$$\begin{array}{lcl} \mathbf{G}_b[\![g_1; g_2]\!] B\ b & = & \text{let } I = \mathtt{in}(g_2) \cup \mathsf{forward}(g_2) \text{ in} \\ & & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!] B\ (b \cup I) \text{ in} \\ & & \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!] B_1\ b \text{ in} \\ & & \quad (b_1 \cup b_2, B_2) \qquad \text{otherwise} \end{array}$$

- *if-then-else.* If the *test* goal of an if-then-else fails, then the *else* branch is performed. This means that the input variables as well as forward use variables of the *else* branch are in backward use w.r.t. the *test* goal. Unlike Prolog, an if-then-else does not commit to the first solution of the condition if the condition is met (Henderson, Conway, Somogyi, and Jeffery 1996): if the computation along an execution path covering the *then* branch leads to a failing derivation, then the computation may backtrack to the *else* branch. This can only happen if the *test* goal in the if-then-else is non-deterministic. We consider these two cases separately.

  If the tested goal is deterministic, we define concrete backward use as:

$$\begin{array}{l} \mathbf{G}_b[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!] B\ b \\ \quad = \quad \text{let } I = \mathtt{in}(g_3) \cup \mathsf{forward}(g_3) \text{ in} \\ \qquad \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!] B\ (b \cup I) \text{ in} \\ \qquad \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!] B_1\ b \text{ in} \\ \qquad \text{let } (b_3, B_3) = \mathbf{G}_b[\![g_3]\!] B_2\ b \text{ in} \\ \qquad\quad (b_2 \cup b_3, B_3) \\ \qquad\qquad \text{if } \det(g_1) = \mathtt{det} \text{ or } \mathtt{semidet} \end{array}$$

  Here, the *then* branch is explicitly annotated starting from the initial backward use variables. Indeed, if the *test* goal is deterministic, once the *then* branch is selected, execution can never return to the *else* branch.

  If the test is non-deterministic, then the goal "if g_1 then g_2 else g_3" becomes equivalent to the non-deterministic disjunction "$(g_1, g_2; g_3)$". Developing the semantic rules applicable to this disjunction, we obtain the following

rule:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{G}_b[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!] B\ b} \\
= & \text{let } I = \texttt{in}(g_3) \cup \mathsf{forward}(g_3) \text{ in} & \\
& \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!] B\ (b \cup I) \text{ in} & \\
& \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!] B_1\ b_1 \text{ in} & \\
& \text{let } (b_3, B_3) = \mathbf{G}_b[\![g_3]\!] B_2\ b \text{ in} & \\
& \quad (b_2 \cup b_3, B_3) & \\
& & \text{otherwise}
\end{array}$$

The particularities of this rule are: ① the variables that are input or in forward use to the second branch are considered in backward use when handling the first branch, ② the annotation of g_2, the *then* part, starts with the result of annotating the g_1, as dictated by the semantic rule for conjunctions, ③ the variables in forward use w.r.t. the if-then-else goal may not be altered within that goal which is in accordance with the rule for non-deterministic disjunctions.

This differentiation adds a form of precision. We could simplify and only use the second semantic rule for all types of if-then-else goals, yet for deterministic tests we know that $b_1 \setminus b \subseteq I$, *i.e.*, the only difference between the resulting b_1 and the initial b is the fact that we have added the input variables of the *else* branch. Also, adding the forward use variables is not be needed when the test is deterministic.

- *negation.* A negation can not introduce new choice points. The set of variables in backward use therefore remains the same. Of course, the negated goal itself must still be annotated.

$$\begin{array}{lll}
\mathbf{G}_b[\![\text{not } g]\!] B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g]\!] B\ b \text{ in} \\
& & \quad (b, B_1)
\end{array}$$

- *literal.* Here we have: $\mathbf{G}_b[\![l]\!] B\ b = \mathbf{L}_b[\![l]\!] B\ b$.

Literals. For $\mathbf{L}_b$, which has a similar signature to $\mathbf{G}_b$ namely

$$\mathbf{L}_b : \mathit{Literal} \rightarrow \mathit{Ann}_b \rightarrow \wp(VI) \rightarrow \wp(VI) \rightarrow \mathit{Ann}_b$$

we differentiate three cases: unifications (or other built-ins), deterministic procedure calls, and non-deterministic procedure calls. The two first cases introduce no new additional data structures in backward use, while the non-deterministic calls do.

As we want to present different possible definitions of the backward use variables contributed by a non-deterministic call, we add one auxiliary functions, namely bu. The function is meant to determine the backward use variables to be

taken into account at the considered program point and to be propagated to all the goals and literals following that program point. Its generic signature is:

$$\mathsf{bu} \quad : \quad \mathit{Literal} \rightarrow \wp(VI) \rightarrow \wp(VI)$$

The idea is that this function takes a given literal as argument as well as a set of variables representing the backward use variables due to the previous goals considered in the procedure definition, and returns a new set of backward use variables.

The generic semantic rule defining literals is then:

$$\begin{array}{lcl}
\mathbf{L}_b[\![\text{unif}]\!]B\ b & = & (b, B[pp(\text{unif}), b]) \\
\mathbf{L}_b[\![p]\!]B\ b & = & (b, B[pp(p), b]) \quad \text{if } \mathtt{det}(p) = \mathtt{det} \text{ or } \mathtt{semidet} \\
\mathbf{L}_b[\![p]\!]B\ b & = & (\mathsf{bu}(p, b), B[pp(p), \mathsf{bu}(p, b)]) \qquad \text{otherwise}
\end{array}$$

Entire Program. Finally, the annotation of the entire program consists of a straightforward annotation of each of the procedures defined in it.

Figure 7.2 recapitulates the entire backward use information defined on the domain of concrete sets of data structures. Figure 7.1 shows the signatures of the semantic functions.

$$\begin{array}{lcl}
\mathit{Ann}_b & = & \mathsf{pp} \rightarrow \wp(VI) \\
\mathbf{Pr}_b & : & \mathit{Procedure} \rightarrow \mathit{Ann}_b \rightarrow \mathit{Ann}_b \\
\mathbf{G}_b & : & \mathit{Goal} \rightarrow \mathit{Ann}_b \rightarrow \wp(VI) \rightarrow \wp(VI) \rightarrow \mathit{Ann}_b \\
\mathbf{L}_b & : & \mathit{Literal} \rightarrow \mathit{Ann}_b \rightarrow \wp(VI) \rightarrow \wp(VI) \rightarrow \mathit{Ann}_b
\end{array}$$

Figure 7.1: Signatures of the semantic function defining $\mathcal{S}em_b$.

7.3.2 Instantiations for bu

We present two simple instantiations of the auxiliary function bu used to approximate the variables that may still be needed upon backtracking due to non deterministic procedure calls.

Instantiation 1. A first rough instantiation of bu is to consider that all the arguments of the called procedure, as well as all the forward use variables at that program point, are added to the already existing set of backward use variables. This is indeed safe as it guarantees that whenever execution comes back to the current literal, all the data structures involved with that literal, as well as the data structures that will again be accessed by the goals following that literal, will remain untouched w.r.t. structure reuse. Thus, in this case we have:

$$\mathsf{bu}^1(p, b) \quad = \quad b \cup \mathsf{forward}(p) \cup \mathit{Vars}(p)$$

Instantiation 2. Obviously, the previous instantiation is a real over-approximation of what really is accessed upon backtracking, and may therefore severely limit the possibilities of reuse in the presence of non-deterministic calls, not only for the literals following the considered literal, but also for the considered literal itself. The above definition can be refined by arguing that explicitly adding the variables of the procedure call should not be needed. Differentiating the output variables from the input variables of p, the non-deterministic procedure call, we may reason as follows:

- The output variables of p, the non-deterministic procedure call, will usually already be in forward use, and if some of these output variables are not in forward use, then this means that the procedure call is free to do whatever it likes to do with these arguments as they are not needed anyway in the calling environment.
- If input variables are still needed after the call to p, then they will belong to the set of variables in forward use or backward use (depending on whether these variables are needed by forward or backward execution), and will therefore already be accounted for. On the other hand, if an input variable is not in use at that program point, then this means that the current call is the last call manipulating that variable directly, hence, if the situation occurs, p might as well allow structure reuse on the data structure pointed at by that variable, and thus the reuse of that structure becomes a local concern of p.

The instantiated function will then look as follows:

$$\mathsf{bu}^2(p, b) \quad = \quad b \cup \mathsf{forward}(p)$$

Obviously, this instantiation, unlike the previous one, allows p to reuse parts of its own input as long as this input is not used by any other literal in forward or backward execution.

Example 7.4 *We derive backward use information for the non-deterministic variant of the recursive procedure* append*. Figure 7.3 recalls the definition of this procedure. This figure explicitly shows the program points and also names each of the goals.*

For each goal g we explicitly list the backward use information available at the moment when that goal is considered (b_g), and the resulting backward use as a result of the semantic function for that goal (b'_g). The program point annotations are represented as b_i, for $(1) \leq i \leq (5)$. The results are shown in Figure 7.4. The final program point annotations, relative to the bu *instantiations used, are recapitulated in Figure 7.5.*

We use extra indentation to make the nesting of the goals explicit.

It is interesting to see that using the second instantiation, neither Y nor Zs are considered to be in backward use at program points (4) and (5). Indeed, Y is output of the append *procedure. Therefore, if that argument is not used in the callers context of*

append, *there is no reason to keep that argument alive, and therefore protect it against possible reuses. Argument Zs is an input argument for the recursive call, yet it is neither in forward use, nor in backward use at that program point, hence, the recursive call is free to reuse parts of Zs if need be.*

Example 7.5 *The resulting backward use annotations for the program given in Example 7.1 is recapitulated in the following table:*

pp	forward(pp)	$Inst_1$	$Inst_2$
1	$\{F, R\}$	$\{\}$	$\{\}$
2	$\{\}$	$\{R\}$	$\{R\}$
3	$\{\}$	$\{E, R\}$	$\{\}$
4	$\{LL\}$	$\{\}$	$\{\}$
5	$\{\}$	$\{X, LL\}$	$\{\}$

Using either Instantiation 1 or Instantiation 2, in both cases, variable R is considered to be in backward use at program point (2)*. This is due to the non-deterministic disjunction, where all input variables to the second branch are added to the backward use variables for the first branch.*

The difference between both instantiations becomes apparent at the procedure calls. In the first instantiation, the literal is annotated with a backward use set including all the arguments of the procedure call, resulting in the sets $\{E, R\}$ *at* (3)*, and* $\{X, LL\}$ *at* (5)*. Obviously, this is an overestimation, which can be avoided using Instantiation 2. In this setting, no variables are considered to be in backward use at these non-deterministic calls. This may seem odd, yet it can be explained using the same reasoning as we applied for variables Y and Zs in the previous example of* append*. Indeed, the inputs are not in backward use as the calls are not followed by any literals using these inputs, similarly for the outputs. In case of the outputs, as they are output arguments of the procedure definitions, it is up to the callers of these procedures to use the computed outputs. If the outputs are not used, then of course, the current procedure may reuse the associated data structures if need be.*

7.4 Analysis Based Backward Use

The aspect that makes the previous approach *simple* is the fact that we describe the backward use information of a non-deterministic procedure call using information at the call site only: we started by adding forward use variables as well as the actual arguments of the procedure call (Instantiation 1), and refined it to include only the forward use variables (Instantiation 2). Yet, the actual backward use contribution can be refined even more by looking what parts of the forward use variables are really accessed upon backtracking in that called procedure. This information can be derived by looking into the called procedure, hence obtaining

a true analysis. If used with a description domain that is capable of describing parts of memory cells to which variables may point to, which is exactly what the domain of concrete and abstract data structures represents, a more precise definition of backward use information can be obtained. We illustrate this with the following example.

Example 7.6 *Consider the following procedure definition:*

```
%:- type t(T) --> f(T,T); g(T).
%:- pred sharedBU(t(T), T).
%:- mode sharedBU(in,out) is nondet.
sharedBU(X,E):-
        (1) X => f(T1, T2),
        (
                (2) E := T1
        ;
                (3) E := T2
        ).
some_predicate(...):- ...,
        A => f(A1,A2),
        (i) sharedBU(A,B),
        A1 => ...,
        ... .
```

If backtracking is ever needed over a call to sharedBU*, then only the second component of* X *will ever be accessed,* i.e.*, variable* $T2$*. Unfortunately, variable* $T2$ *is a local variable to* sharedBU*. In order to use the information of* $T2$ *being in backward use in the context of a call to* sharedBU *we must* translate *that information to the head variables of* sharedBU*. We can either do a rough translation by considering that* $T2$ *is related to* X*, hence* X *is in backward use, or we can make use of the structure sharing relation between* $T2$ *and* X*, and therefore only add the data structure pointing to the second component of* X *to be in backward use,* i.e.*,* $X^{(f,2)}$ *in the concrete domain, and* $X^{\overline{(f,2)}}$ *in an abstract setting.*

The result of this finer grained approach to backward use information is that the call sharedBU(A,B) *adds* $A^{(f,2)}$ *(or* $A^{\overline{(f,2)}}$ *in the abstract domain) to the entities in backward use from program point* (i) *on. This is a correct description of the fact that only the second component of the first argument should be* protected *from any reuse, as this component will be accessed upon reentering* sharedBU(A,B)*. Now, if we assume that only* $A1$*, the second component of* A*, is in forward use w.r.t. the call to* sharedBU(A,B)*, we could slim down the set of structures in backward use even more, and obtain the empty set as the resulting contribution of* sharedBU(A,B) *to the structures in backward use, again taking into account all the available structure sharing information. As an end effect, we can detect that the deconstruction of* $A1$ *performs the last access to* $A1$*, hence, reuse of the deconstructed structure may be allowed.*

Looking at our simplified approaches, we would have obtained:

- (Instantiation 1) *Variables A and B are the head variables of the call to* sharedBU*, and variable* $A1$ *is in forward use, hence, we would augment the set of backward use variables with the set* $\{A, B, A1\}$*. This is a clear overestimation of the above results, and the deconstruction of* $A1$ *will never be recognised as a deconstruction allowing subsequent reuse.*

- (Instantiation 2) *Here, only* $A1$ *is added to the set of variables in backward use. This is needed as we do not exactly know what* sharedBU(A,B) *will need upon backtracking. As a result, the analysis will fail to detect that the deconstruction of* $A1$ *generates garbage cells, hence, the reuse of the deconstructed data structure will not be permitted.*

To define this analysis based approach we will need to define a new set of semantic functions. Given the fact that the simplified approach differs from the analysis based approach mainly in the definition of non-deterministic procedure calls, and given the fact that backward use information remains a goal independent property, the resulting semantic functions will be a hybrid form of the goal-independent based semantics with pre-annotation $\mathcal{S}em_{M\bullet p}$ (Section 5.8) and the previously defined simplified backward use derivation $\mathcal{S}em_b$. As illustrated by the previous example, structure sharing information can be of importance to deduce preciser information, hence, backward use analysis could be defined on top of structure sharing analysis. This could be formalised by adding the goal-independent annotation table obtained by the goal-independent structure sharing analysis defined previously to the list of arguments of the semantic functions defining backward use analysis. Obviously, a fixpoint computation is needed to deal with recursive procedures.

We do not formalise the analysis based approach to backward use information here as the gain in precision seems limited. Instead we illustrate the mechanism of that approach by means of an example. We refer the reader to (Bruynooghe, Janssens, and Kågedal 1997) where this analysis based approach was first described, although the authors did not take structure sharing into account.

Example 7.7 *We illustrate the derivation of the abstract backward use information for the non-deterministic procedure of* append *(c.f. Example 7.4). As the basic process of deriving that information is the same as for the simplified definition, the derivation of backward use for* append *will follow the same scheme as depicted in Figure 7.4, with two differences. First the domain of backward use information is now the domain of abstract data structures, therefore, not* Z *is considered in backward use, but* $Z^{\overline{\epsilon}}$*, thus its abstract data structure. The second difference occurs at goal* g_7*,* i.e.*, program point* (4)*, where the recursive non-deterministic procedure call is performed. In a first iteration we have no backward use information of that call available, therefore, this will be approximated by the empty set. Hence,* $\mathsf{bu}(g_7, b_{g_7}, \{\}) = b_{g_7} \cup \mathsf{forward}(g_7) = \{Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}\}$*. With this*

information, the resulting backward use information of the whole procedure is now:

$$\begin{aligned} b'_{g_0} &= \{Z^{\overline{\epsilon}}\} \cup \{Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}\} \\ &= \{Z^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}\} \end{aligned}$$

Translated (using structure sharing information) and projected onto the input variables of the append *procedure, this yields the singleton set* $\{Z^{\overline{\epsilon}}\}$ *(as both* $Xe^{\overline{\epsilon}}$ *as well as the elements in Xs are related to the elements of Z, and the elements of Z being automatically accounted for in the abstract data structure* $Z^{\overline{\epsilon}}$*) as final backward use set for that procedure.*

After a second iteration, a fixed point is reached.

Clearly, in this example, an analysis based approach yields the same results as our simplified definition. Intuitively, we therefore consider that an analysis based approach to backward use information would be an overkill for the gain of precision it would bring compared to the simplified definition.

7.5 Related Work and Conclusion

In this chapter we formally defined the notions of forward use and backward use. The former expresses the set of variables that may be accessed beyond a certain program point due to forward execution, while the latter collects the set of variables (or data structures) that are accessed upon backtracking over a certain program point.

In (Mulkers 1991) only forward use variables are considered for computing liveness information. The author justifies this restriction by assuming that the run-time system is enhanced in the appropriate way such that whenever data structures are overwritten, a copy of the old values is kept. As we do not rely on such an extension of the run-time system, collecting backward use information is essential.

Forward use and backward use were both formulated in (Bruynooghe, Janssens, and Kågedal 1997). Given our slight redefinition of the language using single procedure goals instead of sets of clauses, we redefined both concepts in the current Mercury syntax. Moreover, we gave some simplified definitions for backward use w.r.t. (Bruynooghe, Janssens, and Kågedal 1997), and argued that the precision obtained with these simplified views are acceptable compared to the analysis based definition.

$$
\begin{array}{lcll}
\mathbf{Pr}_b[\![h \leftarrow g]\!]B & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g]\!]B\ \mathsf{init}_b \text{ in} & \\
 & & \quad B_1 & \\
\mathbf{G}_b[\![g_1, g_2]\!]B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!]B\ b \text{ in} & \\
 & & \quad \mathbf{G}_b[\![g_2]\!]B_1\ b_1 & \\
\mathbf{G}_b[\![g_1; g_2]\!]B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!]B\ b \text{ in} & \\
 & & \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!]B_1\ b \text{ in} & \\
 & & \quad (b_1 \cup b_2, B_2) & \\
 & & & \text{if } \mathtt{switch}((g_1; g_2)) = \mathsf{true} \\
\mathbf{G}_b[\![g_1; g_2]\!]B\ b & = & \text{let } I = \mathtt{in}(g_2) \cup \mathsf{forward}(g_2) \text{ in} & \\
 & & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!]B\ (b \cup I) \text{ in} & \\
 & & \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!]B_1\ b \text{ in} & \\
 & & \quad (b_1 \cup b_2, B_2) & \text{otherwise} \\
\mathbf{G}_b[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!]B\ b & & & \\
 & = & \text{let } I = \mathtt{in}(g_3) \cup \mathsf{forward}(g_3) \text{ in} & \\
 & & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!]B\ (b \cup I) \text{ in} & \\
 & & \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!]B_1\ b \text{ in} & \\
 & & \text{let } (b_3, B_3) = \mathbf{G}_b[\![g_3]\!]B_2\ b \text{ in} & \\
 & & \quad (b_2 \cup b_3, B_3) & \\
 & & & \text{if } \det(g_1) = \mathtt{det} \text{ or } \mathtt{semidet} \\
\mathbf{G}_b[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!]B\ b & & & \\
 & = & \text{let } I = \mathtt{in}(g_3) \cup \mathsf{forward}(g_3) \text{ in} & \\
 & & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g_1]\!]B\ (b \cup I) \text{ in} & \\
 & & \text{let } (b_2, B_2) = \mathbf{G}_b[\![g_2]\!]B_1\ b_1 \text{ in} & \\
 & & \text{let } (b_3, B_3) = \mathbf{G}_b[\![g_3]\!]B_2\ b \text{ in} & \\
 & & \text{let } g = \text{if } g_1 \text{ then } g_2 \text{ else } g_3 \text{ in} & \\
 & & \quad (b_2 \cup b_3, B_3) & \\
 & & & \text{otherwise} \\
\mathbf{G}_b[\![\text{not } g]\!]B\ b & = & \text{let } (b_1, B_1) = \mathbf{G}_b[\![g]\!]B\ b \text{ in} & \\
 & & \quad (b, B_1) & \\
\mathbf{G}_b[\![l]\!]B\ b & = & \mathbf{L}_b[\![l]\!]B\ b & \\
\mathbf{L}_b[\![\text{unif}]\!]B\ b & = & (b, B[pp(\text{unif}), b]) & \\
\mathbf{L}_b[\![p]\!]B\ b & = & (b, B[pp(p), b]) & \text{if } \mathtt{det}(p) = \mathtt{det} \text{ or } \mathtt{semidet} \\
\mathbf{L}_b[\![p]\!]B\ b & = & (\mathsf{bu}(p, b), B[pp(p), \mathsf{bu}(p, b)]) & \text{otherwise}
\end{array}
$$

Figure 7.2: Simple backward use information, c.f. Section 7.3.2 for the function $\mathsf{bu}(p, b)$.

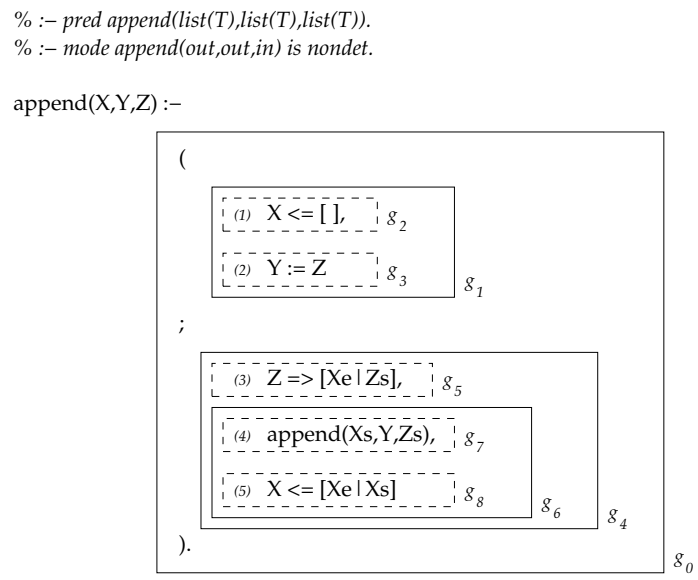


Figure 7.3: Non-deterministic version of the list concatenation predicate append. The goals are explicitly named.

$$
\begin{array}{l}
g_0 \left[\begin{array}{ll}
b_{g_0} = \mathsf{init}_b = \{\,\} & \\
g_1 \left[\begin{array}{ll}
b_{g_1} = b_{g_0} \cup \mathtt{in}(g4) = \{\,\} \cup \{Z\} = \{Z\} & \\
g_2 \left[\begin{array}{ll} b_{g_2} = b_{g_1} = \{Z\} \\ b_{(1)} = b_{g_2} = \{Z\} \\ b'_{g_2} = b_{g_2} = \{Z\} \end{array}\right. & \\
g_3 \left[\begin{array}{ll} b_{g_3} = b'_{g_2} = \{Z\} \\ b_{(2)} = b_{g_3} = \{Z\} \\ b'_{g_3} = b_{g_3} = \{Z\} \end{array}\right. & \\
b'_{g_1} = b'_{g_3} = \{Z\} &
\end{array}\right. & \\
g_4 \left[\begin{array}{ll}
b_{g_4} = b_{g_0} = \{\,\} & \\
g_5 \left[\begin{array}{ll} b_{g_5} = b_{g_4} = \{\,\} \\ b_{(3)} = b_{g_5} = \{\,\} \\ b'_{g_5} = b_{g_5} = \{\,\} \end{array}\right. & \\
g_6 \left[\begin{array}{ll}
b_{g_6} = b'_{g_5} = \{\,\} & \\
g_7 \left[\begin{array}{ll} b_{g_7} = b_{g_6} = \{\,\} \\ b_{(4)} = \mathsf{bu}(g_7, b_{g_7}) = \mathsf{bu}(g_7, \{\,\}) \\ b'_7 = \mathsf{bu}(g_7, b_{g_7}) = \mathsf{bu}(g_7, \{\,\}) \end{array}\right. & \\
g_8 \left[\begin{array}{ll} b_{g_8} = b'_{g_7} = \mathsf{bu}(g_7, \{\,\}) \\ b_{(5)} = b_{g_8} = \mathsf{bu}(g_7, \{\,\}) \\ b'_{g_8} = b_{g_8} = \mathsf{bu}(g_7, \{\,\}) \end{array}\right. & \\
b'_{g_6} = b'_{g_8} = \mathsf{bu}(g_7, \{\,\}) &
\end{array}\right. & \\
b'_{g_4} = b'_{g_6} = \mathsf{bu}(g_7, \{\,\}) &
\end{array}\right. & \\
b'_{g_0} = b'_{g_1} \cup b'_{g_4} = \{Z\} \cup \mathsf{bu}(g_7, \{\,\}) &
\end{array}\right.
\end{array}
$$

Figure 7.4: Backward use annotations for the non-deterministic procedure append defined in Figure 7.3.

pp	forward(pp)	b_{pp}	Inst_1	Inst_2
(1)	$\{Z\}$	$\{Z\}$	$\{Z\}$	$\{Z\}$
(2)	$\{\,\}$	$\{Z\}$	$\{Z\}$	$\{Z\}$
(3)	$\{Xe, Zs\}$	$\{\,\}$	$\{\,\}$	$\{\,\}$
(4)	$\{Xe, Xs\}$	$\mathsf{bu}(g_7, \{\,\})$	$\{Xe, Xs\} \cup \{Xs, Y, Zs\}$ $= \{Xe, Xs, Y, Zs\}$	$\{Xe, Xs\}$
(5)	$\{\,\}$	$\mathsf{bu}(g_7, \{\,\})$	$\{Xe, Xs\} \cup \{Xs, Y, Zs\}$ $= \{Xe, Xs, Y, Zs\}$	$\{Xe, Xs\}$

Figure 7.5: Summary of the backward use annotations for the non-deterministic procedure append for each of the possible instantiations of bu, c.f. Example 7.4.

Chapter 8

Liveness Information

This chapter defines the notion of liveness in the concrete domain, as well as in the abstract domain. If a term on the heap is not live, then it is garbage. The way we deal with this garbage is presented in the chapters that follow.

8.1 Introduction

Using the terminology of data structure introduced in the previous chapters, a data structure is considered *live* at a given literal encountered during the execution of a particular program, if that data structure is accessed by a literal evaluated *after* that given literal within the same program execution.

Example 8.1 *Consider the following fragment of program code showing a sketch of a procedure definition, where Y is of type* `list(int)`*:*

```
p(Y, ...) :-
    % ...
    (i) Y => [First|Rest],
    % ...
```

Consider the query (1) `X = [1,2,3]`, (2) `p(X,...)`, (3) `q(...)`*, with q a call to some other procedure.*

In program point (2) *we can see that p has a unique reference to the heap cells pointed at by X if X does not appear as an argument in the call of q. Let us consider that this is the case, then we proceed our reasoning within the procedure definition of p, after X being renamed to Y. At program point* (i) *we may wonder whether the deconstruction has the last access to the heap cells used to represent the outermost functor of Y^{ϵ}. This is the case if and only if*

- *Y is not live in the context in which p was called,*

- *Y or some other variable sharing the data structure with Y does not appear in any of the literals following the literal at program point (i), not including the literal at (i) itself,*
- *and there is no non-deterministic code prior to program point (i) involving Y or any of its data structures.*

If all these conditions are met, then we can safely decide that the deconstruction has a unique reference to the memory representing the term to which Y is bound. The deconstruction creates two new references to parts of that term, while the memory representing the immediate list-cell to which Y points becomes garbage.

It is clear from the above example that liveness information in a program point (i) within a procedure p depends

- on the liveness information of the procedure call to p, *i.e.*, those structures that are used and referenced at in the context of the call to p – *global liveness*[1],
- on the syntactic occurrence of variables after that program point within the procedure definition of p – *forward use*,
- it also depends on the occurrence of variables in alternative branches of the literal at program point (i) – *backward use*.
- and finally, on structure sharing information, which can be derived by the sharing analysis presented in the previous chapter,

In this chapter we give a formal definition of liveness information depending on this information. The domain of expressing liveness information is the domain of concrete and abstract data structures. We first detail the ordering and operations on these domains.

8.2 Data Structures as Lattices

We already defined the notions of concrete and abstract data structures in Chapter 6.

8.2.1 Concrete Data Structures

In analogy with the sharing domain where we defined the notions of sharing sets and collecting sharing sets, we define data structure sets, and collecting data structure sets. Let VI be the set of variables of interest, and $\mathcal{D}_{VI}$ the set of context-free data structures.

[1] This automatically includes global forward use and global backward use variables.

Definition 8.1 (Data Structure Set) *A* data structure set *is a tuple* $\langle e, D\rangle$ *where* $e \in Eqn^+$ *and* $D \in \wp(\mathcal{D}_{VI})$.

Definition 8.2 (Valid Data Structure Set) *A data structure set* $\langle e, D\rangle$ *is said to be* valid *if each of the data structures appearing in* D *is valid w.r.t.* e, i.e., $\forall d \in D : \langle e, d\rangle$ *is valid.*

Example 8.2 *In a context described by the constraint* $e = (X = [1,2,3] \wedge Y = [])$, $\langle e, \{X^\epsilon, X^{([|],2)\cdot([|],2)}, Y^\epsilon\}\rangle$ *is valid.*

Mercury is a logic language, hence allowing multiple solutions and multiple inputs. This can be modelled using *collecting data structure sets*.

Definition 8.3 (Collecting Data Structure Set) *Sets of data structure sets are called* collecting data structure sets. *The domain of collecting data structure sets is denoted by* $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$.

Definition 8.4 (Valid Collecting Data Structure Set) *A collecting data structure set is valid iff each of its data structure sets is valid.*

Just like we did for the aliasing information, we restrict our domain of interest to the valid collecting data structure sets.

Example 8.3 *Let* X *be of type* ex *defined in Example 6.1. Then the set of data structures* $\langle e, \{X^\epsilon, X^{(a,1)}, X^{(a,2)}\}\rangle$ *is valid for each constraint* e *in which* X *is bound to a term with outermost functor* $a/2$. *The set* $\langle e, \{X^{(a,1)}, X^{(b,1)}\}\rangle$ *can never be valid as* $(a,1) \not\bowtie (b,1)$ *which means that no ex-equation can model the fact that* X *is bound to a term with functor* $a/2$ *as well as to a term with functor* $b/1$.

Given a set of variables, we use the operation data to denote the set of context-free data structures pointed at by these variables

$$\begin{array}{lcl} \mathsf{data} & : & \wp(VI) \to \wp(\mathcal{D}_{VI}) \\ \mathsf{data}(V) & = & \{D^e \mid D \in V\} \end{array}$$

We define the termshift operation on collecting data structure sets.

Definition 8.5 (Termshift) *Let* $\langle e, D\rangle \in \langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle$, *then:*

$$\mathsf{termshift}(\langle e, D\rangle) = \{X^{s_X \bullet s} \mid X^{s_X} \in D, \exists \tau_X \in \mathcal{T}(\mathcal{V}, \Sigma) : e \models X = \tau_X, s_X \bullet s \in \mathcal{T}_{\tau_X}\}$$

Let $D \in \wp(\mathcal{D}_{VI})$ *be a context-free data structure set, then we have:*

$$\mathsf{termshift}(D) = \{X^{s_X \bullet s} \mid X^{s_X} \in D, s_X \bullet s \in \mathcal{TT}_{\mathtt{type}(X)}\}$$

The previous definitions are naturally extended to collecting data structure sets, i.e., *for* $EDS \in \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$*:*

$$\mathsf{termshift}(EDS) \quad = \quad \{\mathsf{termshift}(ED) \mid ED \in EDS\}$$

The ordering within the domain of data structure sets and collecting data structure sets is as follows.

We define the ordering for context-free data structures as follows:

Definition 8.6 (Ordering in $\wp(\mathcal{D}_{VI})$) *Let* $d \in \mathcal{D}_{VI}$*, and* $D, D_1, D_2 \in \wp(\mathcal{D}_{VI})$*, then the concrete data structure* d *is* directly subsumed *by the set of data structures* D*, iff* $d \in D$*.*

d *is* subsumed *by* D *iff is it directly subsumed by* $\mathsf{termshift}(D)$*. This is denoted as* $d \preceq_{cd} D$*.*

D_1 *is* subsumed *by* D_2*, denoted by* $D_1 \subseteq_{cd} D_2$*, iff, each of the individual data structures described by the first data structure set is subsumed by the second:* $\forall d \in D_1 : d \preceq_{cd} D_2$*.*

The least upper bound operation for context-free data structure sets is simply their union.

Sets of context-free data structures are *equivalent* if they are mutually subsumed, *i.e.*, $D_1 \subseteq_{cd} D_2$ and $D_2 \subseteq_{cd} D_1$, with $D_1, D_2 \in \wp(\mathcal{D}_{VI})$. Consequently, two data structure sets $\langle e_1, D_1\rangle$ and $\langle e_2, D_2\rangle$, are equivalent if e_1 and e_2 are equivalent, and if D_1 and D_2 are equivalent. Equivalence between data structure sets is denoted as $ED_1 \sim ED_2$.

Definition 8.7 (Ordering in $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$) *Collecting data structure sets are ordered by the set-inclusion operation, modulo the data structure set equivalence.*

Let $ED \in \langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle$*,* $EDS \in \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$*, then* ED *is* subsumed *by* EDS*, denoted by* $ED \leq_{cd} EDS$*, iff* $ED \in EDS$ *modulo the equivalence relation* $\sim$*.*

Let $EDS_1, EDS_2 \in \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$*, then* EDS_1 *is* subsumed *by* EDS_2*, denoted with* $EDS_1 \sqsubseteq_{cd} EDS_2$*, iff each of the data structure sets in* EDS_1 *is subsumed by* EDS_2*:* $\forall ED \in EDS_1 : ED \leq_{cd} EDS_2$*.*

The least upper bound of two collecting sets is the union of these sets, denoted with the symbol $\sqcup_{cd}$*.*

The domain $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$, ordered by $\sqsubseteq_{cd}$ is a complete lattice with bottom element $\{\,\}$, and top element the set of all valid data structure sets, *i.e.*, $\{\langle e, D\rangle \mid e \in Eqn^+, D \in \mathcal{D}_{VI}, \langle e, D\rangle \text{ is valid}\}$.

Given the fact that we will need structure sharing information in combination with information about simple data structures, we define an operation with which a property about one single data structure can be *extended* to all the data structures sharing the same set of heap cells it designates. This operation is denoted as extend and is defined as follows:

Definition 8.8 (extend**)** *The signature of* extend *is:*

$$\mathsf{extend} : \wp(\mathcal{D}_{VI}) \to \wp(\mathcal{SD}_{VI}) \to \wp(\mathcal{D}_{VI})$$

and its high-level definition is given by:

$$\begin{array}{l}\mathsf{extend}(D, C)\\ \quad = \quad D \cup \{X^{s_X} \mid Y^{s_Y} \preceq_{cd} D, (X^{s_X} - Y^{s_Y}) \preceq_c C\}\end{array}$$

The intuition behind that operation is to obtain the full set of data structures sharing a common pool of heap cells represented by the original set of data structures: if a property holds for the heap cells pointed at by the data structures in D, then this property will also hold for the data structures in $\mathsf{extend}(D, C)$ in the presence of the concrete sharing set C as they point to the same heap cells.

The following lemma shows that the definition for extend can be simplified by partially using plain set-inclusion operations instead of subsumption relations:

Lemma 8.1 *For all $D \in \wp(\mathcal{D}_{VI})$ and for all $C \in \wp(\mathcal{SD}_{VI})$:*

$$\begin{array}{lll}\mathsf{extend}(D, C) & & \\ & & D \\ = & \cup & \{X^{s_X} \mid Y^{s_Y} \in D, \exists s\,.\,(Y^{s_Y \bullet s} - X^{s_X}) \preceq_c C\} \\ & \cup & \left\{X^{s_X \bullet s} \;\middle|\; Y^{s_Y} \in D, \exists s\,.\,s_Y = s'_Y \bullet s, \left(Y^{s'_Y} - X^{s_X}\right) \preceq_c C\right\}\end{array}$$

Proof The proof is straightforward. It suffices to unfold the definition of subsumption: $Y^{s_Y} \preceq_{cd} D \Leftrightarrow Y^{s_Y} \in D$ or $\exists s\,.\,Y^{s_Y \bullet s} \in D$, and $(X^{s_X} - Y^{s_Y}) \preceq_c C \Leftrightarrow (X^{s_X} - Y^{s_Y}) \in C$ or $\exists s\,.\,s_X = s'_X \bullet s, s_Y = s'_Y \bullet s, \left(X^{s'_X} - Y^{s'_Y}\right) \in C$. By combining each of the possible situations, we obtain the given lemma. □

We overload the extend operation to data structures and sharing sets with context, and to sets of data structures and structure sharing sets. Let $e \in Eqn^+$, $D \in \wp(\mathcal{D}_{VI})$, $C \in \wp(\mathcal{SD}_{VI})$, $EDS \in \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$, $EC \in \langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle$, and $ECS \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, then

$$\begin{array}{lcl}\mathsf{extend}(\langle e, D\rangle, \langle e, C\rangle) & = & \langle e, \mathsf{extend}(D, C)\rangle \\ \mathsf{extend}(\langle e, D\rangle, ECS) & = & \{\mathsf{extend}(\langle e, D\rangle, , EC) \mid EC \in ECS\} \\ \mathsf{extend}(EDS, EC) & = & \{\mathsf{extend}(ED, EC) \mid ED \in EDS\} \\ \mathsf{extend}(EDS, ECS) & = & \{\mathsf{extend}(ED, EC) \mid ED \in DS, EC \in CS\}\end{array}$$

Note that extend of a data structure set w.r.t. a structure sharing set is only defined if both sets are defined in the same context e.

Example 8.4 *Consider the variables A and B of type* list(T) *(Example 6.1). Then*

$$\begin{array}{lcl} \mathsf{extend}\left(\{A^{\epsilon}\}, \left\{\left(A^{([],1)} - B^{([],1)}\right)\right\}\right) & = & \left\{A^{\epsilon}, B^{([],1)}\right\} \\ \mathsf{extend}\left(\left\{A^{([],1)}\right\}, \{(A^{\epsilon} - B^{\epsilon})\}\right) & = & \left\{A^{([],1)}, B^{([],1)}\right\} \end{array}$$

and

$$\begin{array}{l} \mathsf{extend}\left(\{\langle e, \{A^{\epsilon}\}\rangle, \langle e, \{B^{\epsilon}\}\rangle\}, \{\langle e, \left\{\left(A^{([],1)} - B^{([],1)}\right)\right\}\rangle, \langle e, \{(A^{\epsilon} - B^{\epsilon})\}\}\rangle\right) \\ \quad = \left\{\langle e, \{A^{\epsilon}, B^{([],1)}\}\rangle, \langle e, \{A^{\epsilon}, B^{\epsilon}\}\rangle, \langle e, \{B^{\epsilon}, A^{([],1)}\}\rangle\right\} \end{array}$$

where $e \in Eqn^{+}$ *such that the given data structures and structure sharing sets are valid.*

Note that extend has the interesting property of being idempotent.

Proposition 8.1 *The* extend *operation is idempotent:*

$$\forall EDS \in \wp(\langle Eqn^{+}, \wp(\mathcal{D}_{VI})\rangle) \text{ and } \forall ECS \in \wp(\langle Eqn^{+}, \wp(\mathcal{SD}_{VI})\rangle)$$

we have:

$$\mathsf{extend}(\mathsf{extend}(EDS, ECS), ECS) = \mathsf{extend}(EDS, ECS)$$

This is in accordance with the meaning we want to associate with the concrete extend operation, namely to collect all pointers to heap cells sharing some common property. This property holds due to the transitive closure of the sharing relation.

8.2.2 Abstract Data Structures

The abstract counterpart for $\wp(\langle Eqn^{+}, \wp(\mathcal{D}_{VI})\rangle)$ is the set of abstract data structures $\wp(\overline{\mathcal{D}}_{VI})$.

In analogy with the concrete domain, we provide a function data_a that collects the abstract data structures for a given set of variables.

$$\begin{array}{lcl} \mathsf{data}_a & : & \wp(VI) \to \wp(\overline{\mathcal{D}}_{VI}) \\ \mathsf{data}_a(V) = \{D^{\overline{\epsilon}} \mid D \in V\} & & \end{array}$$

We also define a termshift operation:

Definition 8.9 (Termshift) *Let* $AD \in \wp(\overline{\mathcal{D}}_{VI})$*, then*

$$\mathsf{termshift}(AD) = \{X^{\overline{s_X \bullet s}} \mid X^{\overline{s_X}} \in AD, \overline{s_X \bullet s} \in \mathcal{TG}_{\mathtt{type}(X)}\}$$

and the ordering relation which is again defined in terms of the termshift operation:

Definition 8.10 (Ordering in $\wp(\overline{\mathcal{D}}_{VI})$) *Let $a \in \overline{\mathcal{D}}_{VI}$, $A \in \wp(\overline{\mathcal{D}}_{VI})$, then the abstract data structure a is* subsumed *by A, denoted by $a <_{ad} A$, iff $a \in \mathsf{termshift}(A)$.*

Let $A_1, A_2 \in \wp(\overline{\mathcal{D}}_{VI})$, then A_1 is subsumed *by A_2, denoted by $A_1 \sqsubseteq_{ad} A_2$, iff each of the abstract data structures in A_1 is subsumed by A_2: $\forall a \in A_1 : a <_{ad} A_2$.*

The least upper bound is again the set-union operation, here denoted by the symbol $\sqcup_{ad}$.

The domain $\langle \wp(\overline{\mathcal{D}}_{VI}), \sqsubseteq_{ad} \rangle$ is a complete lattice with bottom element $\{\,\}$ and top element $\overline{\mathcal{D}}_{VI}$.

Definition 8.11 (Abstract Extend) *The abstract counterpart for* extend*, denoted by* extend_a*, is defined as follows:*

$$\begin{array}{lcl} \mathsf{extend}_a & : & \wp(\overline{\mathcal{D}}_{VI}) \to \wp(\overline{\mathcal{SD}}_{VI}) \to \wp(\overline{\mathcal{D}}_{VI}) \\ \mathsf{extend}_a(A, AS) & = & A \sqcup_{ad} \left\{ X^{\overline{s_X}} \;\middle|\; Y^{\overline{s_Y}} <_{ad} A, \left(Y^{\overline{s_Y}} - X^{\overline{s_X}}\right) \leq_a AS \right\} \end{array}$$

This definition is similar to the definition of extend for concrete data structures and concrete sharing sets. In analogy to the concrete extend we give the more detailed version for extend_a:

Lemma 8.2 *For all $A \in \overline{\mathcal{D}}_{VI}$ and for all $AS \in \wp(\overline{\mathcal{SD}}_{VI})$:*

$$\begin{array}{l} \mathsf{extend}_a(A, AS) \\ = \quad \begin{array}{ll} & A \\ \sqcup_{ad} & \left\{ X^{\overline{s_X}} \;\middle|\; Y^{\overline{s_Y}} \in A, \exists s\,.\, \left(Y^{\overline{s_Y \bullet s}} - X^{\overline{s_X}}\right) \leq_a AS \right\} \\ \sqcup_{ad} & \left\{ X^{\overline{s_X \bullet s}} \;\middle|\; Y^{\overline{s_Y}} \in A, \exists s\,.\, \overline{s_Y} = \overline{s'_Y \bullet s}, \left(Y^{\overline{s'_Y}} - X^{\overline{s_X}}\right) \leq_a AS \right\} \end{array} \end{array}$$

Example 8.5 *Consider the variables A and B of type* $\mathsf{list}(\mathsf{T})$ *(Example 6.1). Then*

$$\begin{array}{lcl} \mathsf{extend}_a\left(\{A^{\overline{\epsilon}}\}, \left\{\left(A^{\overline{([],1)}} - B^{\overline{([],1)}}\right)\right\}\right) & = & \left\{A^{\overline{\epsilon}}, B^{\overline{([],1)}}\right\} \\ \mathsf{extend}_a\left(\left\{A^{\overline{([],1)}}\right\}, \left\{\left(A^{\overline{\epsilon}} - B^{\overline{\epsilon}}\right)\right\}\right) & = & \left\{A^{\overline{([],1)}}, B^{\overline{([],1)}}\right\} \end{array}$$

Unlike extend, extend_a is not idempotent. This is illustrated by the following example.

Example 8.6 *Let $A = \{X^{\overline{\epsilon}}\}$ and $AS = \{\left(X^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right), \left(Y^{\overline{\epsilon}} - Z^{\overline{\epsilon}}\right)\}$, then we have:*

$$\begin{array}{lcl} \mathsf{extend}_a(A, AS) & = & \{X^{\overline{\epsilon}}, Y^{\overline{\epsilon}}\} \\ \mathsf{extend}_a(\mathsf{extend}_a(A, AS), AS) & = & \mathsf{extend}_a(\{X^{\overline{\epsilon}}, Y^{\overline{\epsilon}}\}, AS) \\ & = & \{X^{\overline{\epsilon}}, Y^{\overline{\epsilon}}, Z^{\overline{\epsilon}}\} \end{array}$$

We define the concretisation function mapping abstract data structures to concrete data structures as follows.

Definition 8.12 (Concretisation of Abstract Data Structures)

$$\begin{array}{lcl} \gamma^{\mathcal{D}} & : & \wp(\overline{\mathcal{D}}_{VI}) \to \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle) \\ \gamma^{\mathcal{D}}(A) & = & \{\langle e, D\rangle \mid X^{s_X} \preceq_{cd} D \Rightarrow X^{\overline{s_X}} <_{ad} A, \langle e, D\rangle \textit{ is valid}\} \end{array}$$

Note that the definition of the concretisation function covers only valid data structure sets.

Lemma 8.3 $(\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle), \gamma^{\mathcal{D}}, \wp(\overline{\mathcal{D}}_{VI}))$ *is an insertion.*

Proof Both $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$ and $\wp(\overline{\mathcal{D}}_{VI})$ are complete lattices. It therefore suffices to show that the concretisation function $\gamma^{\mathcal{D}}$ is both monotonic and co-strict. Monotonicity is naturally implied by the definition of $\gamma^{\mathcal{D}}$, indeed, if $A_1 \sqsubseteq_{ad} A_2$, then clearly all concrete data structures in $\gamma^{\mathcal{D}}(A_1)$ will be present in the concretisation of A_2. The top element in $\wp(\overline{\mathcal{D}}_{VI})$ is $\overline{\mathcal{D}}_{VI}$ which $\gamma^{\mathcal{D}}$ maps onto the set of all valid concrete data structure set in $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$, *i.e.*, $\{\langle e, D\rangle \mid e \in Eqn^+, D \in \mathcal{D}_{VI}, \langle e, D\rangle \text{ is valid}\}$. This corresponds to the top element in $\wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$, and thus $\gamma^{\mathcal{D}}$ is co-strict.

□

We show that $\mathsf{extend}_a \propto \mathsf{extend}$.

Lemma 8.4 extend_a *is a safe approximation for* extend*:* $\mathsf{extend}_a \propto \mathsf{extend}$.

Proof Let $ECS \in \wp(\langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle)$, $ECS \in \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$, $A \in \wp(\overline{\mathcal{D}}_{VI})$ and $AS \in \wp(\overline{\mathcal{SD}}_{VI})$, then we need to prove that if $EDS \subseteq_{cd} \gamma^{\mathcal{D}}(A)$ and $ECS \sqsubseteq_c \gamma^{\mathcal{S}}(AS)$, then $\mathsf{extend}(EDS, ECS) \subseteq_{cd} \mathsf{extend}_a(A, AS)$. This can be shown by applying the definitions of extend, extend_a and $\gamma^{\mathcal{S}}$.

□

8.3 Concrete Liveness

Throughout the previous chapters and sections we already sketched our intended definition of liveness: a heap cell is considered live at some specific moment during the execution of a program if it is accessed by any of the instructions executed after that specific moment. In Chapter 7 we described that these instructions arise by two different execution mechanisms: either it is an instruction that is part of the same execution path as the considered moment—this was called *forward use*, or the instruction is performed upon backtracking after that moment—so called *backward use*.

Let us consider the following fragment of a program:

```
p(X1,...,Xn) :-
        ...,
        (i) q(Y1,...,Ym),
        ... .
q(A1,...,Am) :- ... .
```

Suppose we run the program that the above fragment is part of. At some point during the execution of that program we encounter a call to procedure p/n. That specific call can be described by a constraint set e and a structure sharing set C (for the moment we assume a non collecting semantics). Moreover, let L_0 be the set of heap cells that are live outside of the call to p expressed as a set of data structures. As p has only access to the variables $X1, \ldots, Xn$, we restrain the call information to these variables, yielding the projected and renamed sets: e_p, C_p and $L_{0,p}$. Let us now investigate how we can determine the set of live heap cells when the program execution reaches program point (i) within that call to p/n. The purpose of liveness information is to determine the set of heap cells that the literal at (i) has last access to. Turning this description the other way around means that heap cells are live if they are reached by any of the instructions that can be executed after the literal at (i). The set of live heap cells thus consists of

- the heap cells that are accessed outside of p/n, thus $L_{0,p}$;
- the heap cells that are instantiated before (i) and used after (i);
- the heap cells that are instantiated before (i) and used after a possible failure of the literal at (i) or any of the literals following (i);

In this work we refer to heap cells by means of data structures, and of course, once a data structure is said to be live, then all the data structures referring to the same piece of memory must be considered live too. Therefore, the structure sharing information available at program point (i) needs to be taken into account too.

According to the above description, liveness is a property of the heap cells that already exist before the literal at (i) is executed. In a formal way, this results in sets of data structures that are always valid w.r.t. the constraint set describing the bindings at that moment. Put differently, data structures can not be alive if they do not exist yet. This poses a problem for correctly dealing procedure calls as illustrated by the following reasoning. Consider a call to procedure q/m at program point (i) in the definition of p/n. Let L_i be the liveness computed at program point (i). As liveness only includes existing data structures, we have: $Vars(L_i) \cap \texttt{out}(q(Y1, \ldots, Ym)) = \{\}$, and thus, during a goal-dependent analysis of q/m, the data structures associated to the output variables are not considered live in the calling context of q/m, and the analysis could erroneously conclude that some parts of these output data structures can be reused. This is clearly not

the behaviour we want. Therefore we refine the description of live data structures as follows. In our example of a call to p/n, again assuming a non collecting semantics setting, the heap cells that are live at program point (i) are

- the heap cells that were already live before the call to p/n, *i.e.*, $L_{0,p}$;
- the heap cells that are already instantiated or become instantiated at (i), and that are used by any of the instructions following the literal at (i);
- the heap cells that are already instantiated at (i), and that may be needed upon backtracking.

In terms of data structures and therefore the pointers to these heap cells, we say that the set of live data structures at a program point i is determined by

- the set of data structures that was live when entering p/n, $L_{0,p}$;
- the set of data structures that correspond to variables that are instantiated after the literal at (i) has been performed and that are used in any of the literals following (i). This corresponds to the variables in *forward use* as defined in Section 7.2;
- the set of data structures that correspond to variables instantiated at (i) has been performed and used upon backtracking. This corresponds to the variables said in *backward use*, c.f. Section 7.3;
- and finally, we want to collect all the data structures pointing to the live cells, therefore we need to extend all of the above liveness sets w.r.t. structure sharing as it exists at program point (i).

This view means that the live data structures are not necessarily valid in the context of the constraints collected at program point (i), yet they will be valid in the context of the constraint set obtained right after having performed the literal at that program point.

It is important to note that while both forward and backward use collect data structures that have become instantiated *after* the literal has been performed, we use the structure sharing relations as they exist *before* the literal is executed. This difference will be essential for the precision of the derived information in the abstract domain. This will be made clear in Section 8.4.

Formally, let $\mathsf{forward}(i)$ and $\mathsf{backward}(i)$ denote the variables in forward use, resp. backward use[2], at program point (i), then let the function live with signature

$$\mathsf{live} : \mathsf{pp} \rightarrow \langle Eqn^{+}, \wp(\mathcal{SD}_{VI})\rangle \rightarrow \wp(\mathcal{D}_{VI}) \rightarrow \wp(\mathcal{D}_{VI})$$

[2]Note that with the analysis based derivation of backward use information, we already obtain sets of data structures instead of pure variables, hence, of course, in that case, no conversion with the data operator is needed.

be defined as follows:

$$\begin{array}{rcl} \mathsf{live}(i, \langle e_i, C_i \rangle, L_0) & = & \text{let } F_i = \mathsf{data}(\mathsf{forward}(i)) \text{ in} \\ & & \text{let } B_i = \mathsf{data}(\mathsf{backward}(i)) \text{ in} \\ & & \quad \langle e_i, \mathsf{extend}(L_0 \cup F_i \cup B_i, C_i) \rangle \end{array} \tag{8.1}$$

then the liveness at program point i, denoted by L_i is computed as

$$L_i = \mathsf{live}(i, \langle e_i, C_i \rangle, L_{0,p})$$

Equation (8.1) shows that liveness information at a specific program point depends on the liveness information with which the procedure was called ($L_{0,p}$), structure sharing information (C_i), and forward and backward use information (F_i and B_i). This definition implies that liveness is a call dependent property as the liveness at a program point depends on the initial liveness $L_{0,p}$ with which the procedure to which that program point belongs was called. Yet liveness information does not rely on liveness values at any of the preceding program points which means that also the initial liveness of a procedure call is purely determined by the forward/backward uses, the structure sharing, and the initial liveness further up the call tree. Note that at the start of the execution of a given program, the set of live heap cells is empty, hence, the initial liveness is always the empty set.
As a consequence, no fixpoint computation is needed for deriving liveness information, once all the underlying information is present. Therefore, one could argue to formalise the liveness derivation process as a fully separate process, that purely depends on a pre-annotated program. Yet, given the fact that concrete liveness information is not context-free (an environment representing the variable bindings is needed to know the nature of the heap cells), and its close connection to structure sharing information, we decide to formalise liveness analysis using the concrete domain of structure sharing as a basis, explicitly augmented with liveness information.

Therefore, just as we augmented the domain of ex-equations with sharing information, we now augment structure sharing information with liveness information represented as a set of data structures. As the liveness information is recomputed at each program point using the initial liveness information of the procedure call, it might seem the most natural to thread the initial liveness set along the program and include it in the current description of each program point, hence using tuples $\langle e, C, L_0, L \rangle$ with $e \in Eqn^+$ — the ex-equation component, $C \in \wp(\mathcal{SD}_{VI})$ — the structure sharing component, $L_0, L \in \wp(\mathcal{D}_{VI})$ — and the initial and current liveness components resp., to describe a single computation state at a specific program point, or in a collecting semantics setting, using sets of such tuples. This could indeed be seen as a viable concrete domain for deriving liveness information as it makes the link between a current liveness description and the initial liveness description on which it depends explicit, leaving

no doubt that that current description stems for that particular initial liveness description. Unfortunately, this domain makes the definition of the semantics of procedure call literals complicated. Indeed, in that definition, the initial liveness components have to be replaced by the current liveness component in order to have a correct call dependent derivation of the called procedures. Yet, for the resulting semantics of the procedure call, the obtained initial liveness components must again be replaced by the initial liveness components with which the procedure call was considered. This is illustrated by the following example. Consider the procedure definition of a procedure $p/2$ where the literal at program point (i) is a procedure call to a procedure $q/2$.

```
%:- pred p(t,t).
%:- mode p(in,out).
p(X,Y) :- ..., (i) q(A,B), (i+1) r(C,D), ... .
```

where t is a type with type tree $\mathcal{TT}_{\mathtt{t}} = \{s_1, s_2, s_3\}$: The call to p/n is described by the collecting liveness set[3]

$$\{\langle e_{p,1}, \{\,\}, \{X^{s_1}\}, \{\,\}\rangle, \{\langle e_{p,2}, \{\,\}, \{X^{s_2}\}, \{\,\}\rangle\}$$

At program point (i) we could have the description:

$$\{\langle e_1, C_1, \{X^{s_1}\}, L_{q,1}\rangle, \{\langle e_2, C_2, \{X^{s_2}\}, L_{q,2}\rangle\}$$

In order to correctly derive the liveness properties for q called in such circumstances, the initial liveness components of the call description to q must be set to the current descriptions, while the current descriptions themselves become irrelevant, hence obtaining the call description:

$$\{\langle e_1, C_1, L_{q,1}, \{\,\}\rangle, \{\langle e_2, C_2, L_{q,2}, \{\,\}\rangle\}$$

Now suppose that this call description yields the exit description[4]

$$\{\langle e_3, C_3, L_{q,1}, \{\,\}\rangle, \{\langle e_4, C_4, L_{q,2}, \{\,\}\rangle\}$$

The only parts of interest of this exit description are the ex-equations and the structure sharing components. If we would continue the analysis using that exit description as a basis to analyse the following literal at program point $(i+1)$, then we would obtain wrong results. Indeed, the liveness at program point $(i+1)$ should depend on the same initial liveness as with which program point (i)

[3] Note that the current liveness description for a specific procedure call is in fact irrelevant, as only the initial liveness component is of importance for the correct derivation of the liveness information within a procedure. At best, the current liveness component could simply be identical to the initial liveness component.

[4] Again, the current liveness descriptions are irrelevant, as they can be recomputed again if needed, yet they don't depend on the internals of the called procedure.

was analysed, and not with the resulting initial liveness after having analysed program point (i). Hence, to be correct, the results of the analysis of the literal at program point (i) should be combined with the call description of that literal as we would need to keep the initial liveness set of the former, and the constraint set as well as the structure sharing set of the latter.
Therefore, using tuples containing the current liveness as well as the initial liveness sets leads to an obfuscated and complicated formalisation which is why we propose the following simplified view.

At the start of the execution of a program, the query that will be performed can be adorned with the description that none of the involved variables are constrained (the ex-equation is therefore simply true), that there is no structure sharing, and obviously, as there are no data structures yet, there can not be any live data structures either. During the execution of the program, each procedure call is also described by tuples consisting of an ex-equation describing the variable bindings and a structure sharing component describing the possible sharing between the data structures of these variables. The liveness information for each procedure call is uniquely determined by the local information available for that call, but, as we argued earlier, it is also determined by the initial liveness of the procedure where the procedure call occurs. In the previous section we opted for threading that extra information along in the current description, yet now we will show that each individual current description consisting of a tuple $\langle e, C, L\rangle$ (with e, C having the obvious meaning, and where L would denote the current liveness information) corresponds to at most one such tuple of the original call description of the procedure to which that tuple belongs, hence, each such tuple can be mapped in a unique way to the original initial liveness information of the procedure call. In order to prove this statement we first give the necessary details of this new domain and define the corresponding semantics.

We use the following definition for *liveness descriptions*:

Definition 8.13 (Liveness Description) *A tuple in* $\langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle$ *is a* liveness description. *Let* $\langle e, C, L\rangle$ *be such a liveness description, then* e *is called the* constraint component, C *the* sharing component *and* L *the* liveness component.

The actual concrete domain is the powerset over liveness descriptions, *i.e.*, $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle)$, which we abbreviate to $\mathcal{CL}$. Elements from this domain are called *collecting liveness descriptions*.

8.3.1 Operations, Ordering

The operations on the domain of structure sharing are naturally adapted to the collecting liveness descriptions: termshift, projection and renaming now include the termshift, projection and renaming resp. of the liveness component.

Definition 8.14 (Ordering in $\langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle$) *Let $cl_1 = \langle e, C_1, L_1\rangle$ and $cl_2 = \langle e, C_2, L_2\rangle$ be two liveness descriptions, then we say that the first is* subsumed *by the second, denoted by $cl_1 \preceq_{cl} cl_2$ iff $C_1 \subseteq_c C_2$, and $L_1 \subseteq_{cd} L_2$.*

Note that the ordering of two liveness descriptions is only defined for equivalent environments.

Two liveness descriptions are considered equivalent if they are mutually subsumed.

The ordering in the power set of the above domain, *i.e.*, $\mathcal{CL}$ is given by the following definition:

Definition 8.15 (Ordering in $\mathcal{CL}$) *Collecting liveness descriptions are ordered by the set-inclusion operation, modulo the equivalence of liveness descriptions.*

Let $cl \in \langle Eqn^+, \wp(\mathcal{SD}_{VI}), \wp(\mathcal{D}_{VI})\rangle$, and $CL \in \mathcal{CL}$, then cl is subsumed *by CL, denoted by $cl \leq_{cl} CL$, iff $cl \in CL$.*

Let $CL_1, CL_2 \in CL$, then CL_1 is subsumed by CL_2, denoted by $CL_1 \sqsubseteq_{cl} CL_2$, iff $\forall cl \in CL_1 : cl \leq_{cl} CL_2$, in other words, $CL_1 \subseteq CL_2$.

Finally, the least upper bound of two concrete collecting liveness descriptions is simply the union of these sets, denoted with the symbol $\sqcup_{cl}$.

As liveness information is of interest at every program point and not only at unification literals as suggested by our usual semantics in which the auxiliary function add is used, we slightly adapt the natural semantics of Mercury programs.

8.3.2 Augmented Natural Semantics

For the clarity of the presentation we consider that forward use as well as backward use information are available at each program point within our Mercury programs. Forward use can be queried by the function forward (Defined in Section 7.2). We assume that backward use information at a program point i is given in a similarly way, *i.e.*, by a function backward : pp $\rightarrow \wp(\mathcal{D}_{VI})$ (c.f. Section 7.3 for the possible interpretations of this function).

The derivation of liveness information follows slightly different rules as for our previous domains, hence, based on the natural semantics $\mathcal{S}em_M$ we present a variation on that semantics, resulting in the semantics $\mathcal{S}em_{M^+}$ with semantic functions $\mathbf{R}_{M^+}$, $\mathbf{P}_{M^+}$, $\mathbf{G}_{M^+}$, etc. The variation lies mainly in the definition of literal goals as this is the only place of interest for recomputing the current liveness. The only purpose of recomputing that liveness is to record it in the annotation table obtained as a result of the semantics and to use the obtained call description for a correct derivation of the procedure calls. Hence, we have the following definition:

$$\mathbf{G}_{M^+}[\![l]\!](e,A)\mathcal{S}_0\mathcal{S} \quad = \quad \text{let } \mathcal{S}_1 = \mathsf{update}(pp(l),\mathcal{S}_0,\mathcal{S}) \text{ in } (\mathbf{L}_{M^+}[\![l]\!]\, e\, \mathcal{S}_1, A[(\mathrm{pp}(l),\mathcal{S}_0),\mathcal{S}_1])$$

using a new auxiliary function update. The intended meaning of update is to update the liveness component of the call description for the current literal taking into account the initial call description of the procedure to which the literal belongs. Leaving the remainder of the semantic rules the same, we automatically obtain that a procedure call is looked at with an updated liveness component, and therefore analysed in a correct way. The resulting exit description of such a procedure call may contain a new liveness component, yet, this liveness component is irrelevant and is never actually used for any other computation.

Obviously, the new set of semantic functions is well defined if all the instantiations of the auxiliary operations are monotonic.

Before defining the concrete liveness semantics, we introduce the following handy operations mapping elements in $\mathcal{CL}$ to liveness descriptions in the domain $\wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle)$ and the other way around:

$$\begin{array}{lcl} \mathsf{reduce} & : & \mathcal{CL} \to \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle) \\ \mathsf{reduce}(CL) & = & \{\langle e, C\rangle \mid \langle e, C, L\rangle \in CL\} \\ \mathsf{augment} & : & \wp(\langle Eqn^+, \wp(\mathcal{SD}_{VI})\rangle) \to \mathcal{CL} \\ \mathsf{augment}(CS) & = & \{\langle e, C, \{\,\}\rangle \mid \langle e, C\rangle \in CS\} \end{array} \tag{8.2}$$

Definition 8.16 (Concrete Liveness Derivation) *We instantiate the augmented natural semantics with our concrete liveness domain $\mathcal{CL}$ where the instantiated auxiliary functions,* $\mathsf{init}^{\mathcal{CL}}, \mathsf{add}^{\mathcal{CL}}, \mathsf{comb}^{\mathcal{CL}}, \mathsf{update}^{\mathcal{CL}}$ *are defined as follows:*

$$\begin{array}{lcl} \mathsf{init}^{\mathcal{CL}} & = & \mathsf{augment}(\mathsf{init}_c) \\ \mathsf{comb}^{\mathcal{CL}}(CL_o, CL_n) & = & \mathsf{augment}(\mathsf{comb}_c(\mathsf{reduce}(CL_o), \mathsf{reduce}(CL_n))) \\ \mathsf{add}^{\mathcal{CL}}(unif, CL) & = & \mathsf{augment}(\mathsf{add}_c(unif, \mathsf{reduce}(CL))) \\ \mathsf{update}^{\mathcal{CL}}(i, CL_o, CL_n) & = & \left\{ \langle e, C, L\rangle \;\middle|\; \begin{array}{l} \langle e_o, C_o, L_o\rangle \in CL_o, \\ \langle e_n, C_n, L_n\rangle \in CL_n, \\ e_o \models e_n, C_o \subseteq_c C_n, \\ e = e_n, C = C_n, \\ L = \mathsf{live}(i, \langle e, C\rangle, L_o) \end{array} \right\} \end{array}$$

where $\mathsf{init}_c, \mathsf{add}_c, \mathsf{comb}_c$ *and* ECS_{unif} *are as defined in Definition 6.24 (page 120).*

We discuss the instantiation of each of the auxiliary functions.

- $\mathsf{init}^{\mathcal{CL}}$. In this definition, $\mathsf{init}^{\mathcal{CL}}$ results in the description $\langle \mathsf{true}, \{\,\}, \{\,\}\rangle$. This description means that the query is executed starting from an empty constraint set, no structure sharing and no current (therefore initial) liveness.

- $\mathsf{comb}^{\mathcal{CL}}$. The main purpose of the combination operation $\mathsf{comb}^{\mathcal{CL}}$ is to correctly combine the structure sharing information. As the liveness components are irrelevant, and are recomputed when needed anyway, we can simply reduce the liveness descriptions to correct sharing descriptions, combine the resulting sharing descriptions, and then augment the result to obtain elements in $\mathcal{CL}$. The result is a set of liveness descriptions with new updated constraint and structure sharing components, yet with re-initialised liveness components.

- $\mathsf{add}^{\mathcal{CL}}$. Adding a unification to a given description is analog to the combination operation as it simply consists of adding the sharing information to the collecting liveness description. This means that we can simply define $\mathsf{add}^{\mathcal{CL}}$ in terms of adding the unification to the structure sharing description obtained from reducing the initial set of liveness descriptions.

- $\mathsf{update}^{\mathcal{CL}}$. The new auxiliary function $\mathsf{update}^{\mathcal{CL}}$ updates the current liveness component by computing the liveness information for each tuple in the description individually, using the liveness component of the liveness description of the call with matching constraint component and matching structure sharing component. We use the live operation introduced earlier (Equation 8.1). Note that the new computed liveness component is independent of the liveness component that was already present in the description.

As a conclusion, it is interesting to see that the liveness component is never actually used to compute further call descriptions, except for the the new call descriptions of procedure calls and for the purpose of being recorded in the annotation table.

We present an interesting property of the liveness information gathered that way.

Definition 8.17 *In a particular liveness call description $\langle e, C, L\rangle$, L is said to be* synchronised *with the structure sharing component C, if $\forall \alpha \in L : (\alpha - \beta) \subseteq_c C \Rightarrow \beta \in L_0$.*

Corollary 8.1 *In the context of Definition 8.16, for each derived liveness description $\langle e, C, L\rangle$ in a collecting call description CL for a procedure call, each L is synchronised with the corresponding sharing component C.*

This property is due to the fact that the liveness component at a literal is always updated using $\mathsf{update}^{\mathcal{CL}}$ (with a structure sharing component that is closed under transitive closure) before performing the procedure call corresponding to that literal.

Neither $\mathsf{comb}^{\mathcal{CL}}$ nor $\mathsf{add}^{\mathcal{CL}}$ are completely monotonic functions (as they both initialise the liveness component of the resulting liveness description), yet all the auxiliary operations are monotonic w.r.t. the structure sharing component. This is simply the effect of liveness information being a local property based on in use and structure sharing information only. Hence, with liveness derivation being a structure sharing driven process, and with the auxiliary operations being monotone in their structure sharing components, we can safely conclude that the resulting semantic functions are therefore monotonic too and thus $\mathcal{S}em_{M^+}(\mathcal{CL})$ is well defined.

We illustrate the previous concepts with the classic example of the deterministic procedure of append.

Example 8.7 *Consider the procedure definition of the deterministic version of* append *of Example 4.4 which we repeat here:*

```
% :- pred append(list(T),list(T),list(T)).
% :- mode append(in,in,out) is det.
append(X,Y,Z) :-
        (
                (1) X == [],
                (2) Z := Y
        ;
                (3) X => [Xe|Xs],
                (4) append(Xs,Y,Zs),
                (5) Z <= [Xe|Zs]
        ).
```

We consider the collecting liveness call description

$$CL_0 = \left\{ \begin{array}{l} \langle \{X = [f(1)], Y = [f(2)]\}, \{\,\}, \{Y^{\epsilon}\} \rangle, \\ \langle \{X = [f(1)], Y = [f(1), f(2)]\}, \{\left(X^{([],1)} - Y^{([],1)}\right)\}, \{Y^{\epsilon}, X^{([],1)}\} \rangle \end{array} \right\}$$

This specifies the bindings of X and Y. In the second description, the element of X is shared with the first element of Y. In both descriptions, Y^{ϵ} is live outside of the call to append*. In the presence of the structure sharing between X and Y, the elements of X must also be live in the calling environment described by the second description (in order to be synchronised with the structure sharing information).*

The descriptions that are obtained for each of the program points are given in Table 8.1.

From this table, we see that at program point (3) *the list cell of the deconstructed variable X is not live, hence may be reused for this specific call.*

1	F_1	$=$	$\{Y\}$
	CL_1	$=$	CL_0
2	F_2	$=$	$\{\,\}$
	CL_2	$=$	$\{\,\}$ — Inconsistent constraint component
3	F_3	$=$	$\{Y, Xe, Xs\}$
	CL_3	$=$	$\left\{ \begin{array}{l} \left\{ \begin{array}{lcl} e & = & \{X = [f(1)], Y = [f(2)]\} \\ C & = & \{\,\} \\ L & = & \boxed{\{Xe^\epsilon, Xs^\epsilon, Y^\epsilon\}} \end{array} \right. \\ \left\{ \begin{array}{lcl} e & = & \{X = [f(1)], Y = [f(1), f(2)]\} \\ C & = & \{\left(X^{([],1)} - Y^{([],1)}\right)\} \\ L & = & \boxed{\{Xe^\epsilon, Xs^\epsilon, Y^\epsilon, X^{([],1)}\}} \end{array} \right. \end{array} \right\}$
4	F_4	$=$	$\{Xe, Zs\}$
	CL_4	$=$	$\left\{ \begin{array}{l} \left\{ \begin{array}{lcl} e & = & \{X = [f(1)], Y = [f(2)], X = [Xe\|Xs]\} \\ C & = & \left\{\left(Xe^\epsilon - X^{([],1)}\right), \left(Xs^\epsilon - X^{([],2)}\right)\right\}, \\ L & = & \boxed{\{Y^\epsilon, Xe^\epsilon, Zs^\epsilon, X^{([],1)}\}} \end{array} \right. \\ \left\{ \begin{array}{lcl} e & = & \{X = [f(1)], Y = [f(1), f(2)], X = [Xe\|Xs]\} \\ C & = & \left\{ \begin{array}{l} \left(X^{([],1)} - Y^{([],1)}\right), \left(Xe^\epsilon - X^{([],1)}\right), \\ \left(Xs^\epsilon - X^{([],2)}\right), \left(Xe^\epsilon - Y^{([],1)}\right) \end{array} \right\} \\ L & = & \boxed{\{Y^\epsilon, Xe^\epsilon, Zs^\epsilon, X^{([],1)}\}} \end{array} \right. \end{array} \right\}$
5	F_5	$=$	$\{\,\}$
	CL_5	$=$	$\left\{ \begin{array}{l} \left\{ \begin{array}{lcl} e & = & \left\{ \begin{array}{l} X = [f(1)], Y = [f(2)], \\ X = [Xe\|Xs], Zs = Y \end{array} \right\} \\ C & = & \left\{\left(Xe^\epsilon - X^{([],1)}\right), \left(Xs^\epsilon - X^{([],2)}\right), (Zs^\epsilon - Y^\epsilon)\right\} \\ L & = & \boxed{\{Y^\epsilon, Zs^\epsilon\}} \end{array} \right. \\ \left\{ \begin{array}{lcl} e & = & \left\{ \begin{array}{l} X = [f(1)], Y = [f(1), f(2)], \\ X = [Xe\|Xs], Zs = Y \end{array} \right\} \\ C & = & \left\{ \begin{array}{l} \left(X^{([],1)} - Y^{([],1)}\right), \left(Xe^\epsilon - X^{([],1)}\right), \\ \left(Xs^\epsilon - X^{([],2)}\right), \left(Xe^\epsilon - Y^{([],1)}\right), \\ (Zs^\epsilon - Y^\epsilon) \end{array} \right\} \\ L & = & \boxed{\{Y^\epsilon, X^{([],1)}, Zs^\epsilon, Xe^\epsilon\}} \end{array} \right. \end{array} \right\}$

Table 8.1: Liveness details for the deterministic append procedure (Example 8.7). The local forward use at each of the program points is explicitly listed as F_i, $1 \leq i \leq 5$. The backward use sets are empty.

8.4 Abstract Liveness

While the liveness information in the concrete domain can be collected using concrete data structures, in the abstract domain we approximate this information using abstract data structures instead. Just as for the concrete domain, in order to compute the live data structures at a given program point within a procedure p, we need: forward use information, backward use information, the live data structures in the call to p, and finally, sharing information that enables us to extend all the known to be live data structures to all the data structures covering the same heap space.

We illustrate the importance of using the structure sharing information as it exists *before* the literal at the considered program point is performed.

Example 8.8 *Recall that the objective of liveness analysis is to collect the set of live data structures such that, based on that information, the compiler can safely decide which data structures may be reused.*

Now consider the simple deconstruction unification $X \Rightarrow [Xe|Xs]$, where X is assumed to be bound to a term of type $\mathsf{list}(\mathsf{int})$*. Suppose that the literal belongs to a deterministic derivation (hence no backward use), that only Xe and Xs are in forward use, and that there is no structure sharing before the unification. Obviously, after the unification, the tail of X will be shared with Xs. In the concrete domain this is described by the tuple $\left(X^{([|],2)} - Xs^{\epsilon}\right)$. Yet in the abstract domain the most precise description of this relation is $\left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}}\right)$, hence the information that only the tail of X is shared is lost.*

When computing the set of live data structures, this loss of information becomes capital. Determining the live data structures in the absence of any structure sharing, we obtain that only the (abstract) data structures $Xe^{\overline{\epsilon}}$ and $Xs^{\overline{\epsilon}}$ are live (due to forward use). This allows us to conclude that the main functor that X points to is not live, and may therefore be reused, which is a perfectly safe conclusion.

Yet, if the structure sharing that the literal creates is taken into account, then also $X^{\overline{\epsilon}}$ will be considered live too.

For consistency, we will over-line each of the entities involved with the abstract domain, to clearly differentiate them from elements of the concrete domain. Let $\overline{A}_i$ be the correct approximation of the structure sharing information at a program point (i) in p, and let $\overline{L}_{0,p}$ be the set of abstract data structures that approximate the live heap cells in the calling context of p, then we can compute an approximation for the live data structures at program point (i) in a similar way as in the concrete domain. Let live_a, with signature $\mathsf{pp} \to \wp(\overline{\mathcal{SD}}_{VI}) \to \wp(\overline{\mathcal{D}}_{VI}) \to \wp(\overline{\mathcal{D}}_{VI})$, be defined as follows:

$$\begin{array}{lcl} \mathsf{live}_a(i, \overline{A}, \overline{L}_0) & = & \mathsf{let}\ \overline{F}_i = \mathsf{data}_a(\mathsf{forward}(i))\ \mathsf{in} \\ & & \mathsf{let}\ \overline{B}_i = \mathsf{data}_a(\mathsf{backward}(i))\ \mathsf{in} \\ & & \quad \mathsf{extend}_a(\overline{L}_0 \sqcup_{ad} \overline{F}_i \sqcup_{ad} \overline{B}_i, \overline{A}) \end{array} \tag{8.3}$$

then the live structure at i are given by: $\overline{L}_i = \mathsf{live}_a(i, \overline{A}_i, \overline{L}_{0,p})$.

Given the extra required information, and in analogy to the concrete domain of liveness descriptions, we augment the domain of abstract structure sharing with one additional component: the liveness component expressed in terms of abstract data structures. Hence, our abstract descriptions are elements from the domain $\langle \wp(\overline{\mathcal{SD}}_{VI}), \wp(\overline{\mathcal{D}}_{VI}) \rangle$, denoted by $\mathcal{AL}$. Elements in $\mathcal{AL}$ are called *abstract liveness descriptions* and usually denoted by AL, AL_1, etc.

8.4.1 Operations, Ordering

We extend the definitions of the operations defined for abstract structure sharing to include the liveness components: termshift, projection and renaming now also take care of the liveness component.

We define the ordering of the elements in $\mathcal{AL}$ using the ordering of the elements of the domains of its components:

Definition 8.18 (Ordering in $\mathcal{AL}$) *Let $AL_1 = \langle \overline{A}_1, \overline{L}_1 \rangle$, and $AL_2 = \langle \overline{A}_2, \overline{L}_2 \rangle$ both be elements in $\mathcal{AL}$, then AL_1 is* subsumed *by AL_2, denoted by $AL_1 \sqsubseteq_{al} AL_2$, iff $\overline{A}_1 \sqsubseteq_a \overline{A}_2$ and $\overline{L}_1 \sqsubseteq_{ad} \overline{L}_2$.*

The least upper bound of elements in $\mathcal{AL}$, denoted by $\sqcup_{al}$, is defined as the abstract liveness description composed of the least upper bound of the components of the given descriptions: $\langle \overline{A}_1, \overline{L}_1 \rangle \sqcup_{al} \langle \overline{A}_2, \overline{L}_2 \rangle = \langle \overline{A}_1 \sqcup_a \overline{A}_2, \overline{L}_1 \sqcup_{ad} \overline{L}_2 \rangle$.

8.4.2 Abstract Instantiation of the Augmented Semantics

In analogy to the concrete domain, we introduce two additional functions mapping abstract liveness sets to abstract sharing sets and vice versa.

$$\begin{array}{lcl} \mathsf{reduce}_a & : & \mathcal{AL} \to \wp(\overline{\mathcal{SD}}_{VI}) \\ \mathsf{reduce}_a(\langle \overline{A}, \overline{L} \rangle) = \overline{A} & & \\ \mathsf{augment}_a & : & \wp(\overline{\mathcal{SD}}_{VI}) \to \mathcal{AL} \\ \mathsf{augment}_a(\overline{A}) & = & \langle \overline{A}, \{\,\}, \rangle \end{array} \tag{8.4}$$

Definition 8.19 (Abstract Liveness Derivation) *We define the abstract liveness semantics to be the augmented semantics $\mathcal{S}em_{M^+}$ instantiated with the domain $\mathcal{AL}$ and the following auxiliary operations:*

$$\begin{array}{lcl} \mathsf{init}^{\mathcal{AL}} & = & \mathsf{augment}_a(\mathsf{init}_a) \\ \mathsf{comb}^{\mathcal{AL}}(AL_0, AL_n) & = & \mathsf{augment}_a(\mathsf{comb}_a(\mathsf{reduce}_a(AL_o), \mathsf{reduce}_a(AL_n))) \\ \mathsf{add}^{\mathcal{AL}}(\mathit{unif}, AL) & = & \mathsf{augment}_a(\mathsf{add}_a(\mathit{unif}, \mathsf{reduce}_a(AL))) \\ \mathsf{update}^{\mathcal{AL}}(i, \langle \overline{A}_0, \overline{L}_0 \rangle, \langle \overline{A}, \overline{L} \rangle) = \langle \overline{A}, \mathsf{live}_a(i, \overline{A}, \overline{L}_0) \rangle & & \end{array}$$

where init_a, comb_a, add_a *and* A_{unif} *are as defined in the abstract sharing semantics (Definition 6.32) and where* live_a *is given by Equation (8.3).*

In the concrete domain we introduced the notion of synchronised liveness information. In the abstract domain, this notion is not present. The difference is due to the fact that structure sharing in the concrete domain is closed under transitive closure, while in the abstract domain, structure sharing is combined using the alternating closure. Hence, in the concrete domain, if a data structure β is live, then so are all the structures that are shared with it. In the abstract domain, β may be live, yet, even in the presence of an explicit abstract structure sharing relation such as $(\beta - \gamma)$, γ does not need to be live. The following example illustrates this.

Example 8.9 *Let* $\overline{A} = \{(\alpha - \beta), (\beta - \gamma)\}$*, in the presence of an initial liveness set* $\overline{L}_0 = \{\}$*, and where the structures in forward or backward use at program point i are given by the set* $\overline{U}_i = \{\alpha\}$*. Then* $\mathsf{update}^{\mathcal{AL}}(i, \langle \overline{A}_0, \overline{L}_0 \rangle, \langle \overline{A}, \overline{L} \rangle) = \langle \overline{A}, \overline{L}' \rangle$ *where* $\overline{A}_0$ *and* $\overline{L}$ *are irrelevant here, and where*

$$\begin{array}{rcl} \overline{L}' & = & \mathsf{live}_a(i, \overline{A}, \overline{L}_0) \\ & = & \mathsf{extend}_a(\overline{L}_0 \sqcup_{ad} \overline{U}_i, \overline{A}) \\ & = & \mathsf{extend}_a(\{\alpha\}, \{(\alpha - \beta), (\beta - \gamma)\}) \\ & = & \{\alpha, \beta\} \end{array}$$

Hence, in the current liveness component, only α *and* β *are live, even in the presence of the structure sharing between* β *and* γ*. In fact, the absence of the explicit structure sharing between* α *and* γ *in* $\overline{A}$ *means that the structure sharing pairs* $(\alpha - \beta)$ *and* $(\beta - \gamma)$ *stem from two different computation paths, hence are not combined with each other.*

8.4.3 Safe approximation

We define the concretisation of abstract liveness descriptions as the composition of the concretisation of each of its components: the concretisation $\gamma^{\mathcal{S}}$ of the sharing component is given in Definition 6.28, and the concretisation $\gamma^{\mathcal{D}}$ of the abstract data structures representing the liveness components is defined in Definition 8.12. As each of these concretisation functions is an insertion w.r.t. their relative concrete and abstract domain (Lemma 6.1, and Lemma 8.3 resp.), the global concretisation function, denoted by $\gamma^{\mathcal{L}}$, is also an insertion between the concrete domain $\mathcal{CL}$ and its abstract counterpart $\mathcal{AL}$. According to Theorem 5.1 it suffices to show that each of the auxiliary operations in $\mathcal{S}em_{M^+}(\mathcal{CL})$ is correctly approximated by the instantiations of these auxiliary operations in $\mathcal{S}em_{M^+}(\mathcal{AL})$. We show this for each of the individual auxiliary functions.

Lemma 8.5 $\mathsf{init}^{\mathcal{AL}} \propto \mathsf{init}^{\mathcal{CL}}$.

Proof $\mathsf{init}^{\mathcal{AL}} = \langle\{\,\},\{\,\}\rangle$. We have $\gamma^{\mathcal{L}}(\mathsf{init}^{\mathcal{AL}}) = \{\langle e,\{\,\},\{\,\}\rangle \mid e \in Eqn^{+}\}$, which clearly subsumes $\mathsf{init}^{\mathcal{CL}} = \{\langle \mathsf{true},\{\,\},\{\,\}\rangle\}$.

□

Lemma 8.6 $\mathsf{comb}^{\mathcal{AL}} \propto \mathsf{comb}^{\mathcal{CL}}$.

Proof Clearly $\mathsf{augment}_a \propto \mathsf{augment}$ and $\mathsf{reduce}_a \propto \mathsf{reduce}$. As $\mathsf{comb}_a \propto \mathsf{comb}_c$ (Lemma 6.5), obviously $\mathsf{comb}^{\mathcal{AL}} \propto \mathsf{comb}^{\mathcal{CL}}$.

□

Lemma 8.7 $\mathsf{add}^{\mathcal{AL}} \propto \mathsf{add}^{\mathcal{CL}}$.

Proof Both $\mathsf{add}^{\mathcal{AL}}$ and $\mathsf{add}^{\mathcal{CL}}$ are defined in terms of $\mathsf{augment}_a$, reduce_a, add_a and augment, reduce add_c resp. With $\mathsf{add}_a \propto \mathsf{add}_c$ (Lemma 6.6) and $\mathsf{augment}_a \propto$ augment, $\mathsf{reduce}_a \propto$ reduce, we obtain $\mathsf{add}^{\mathcal{AL}} \propto \mathsf{add}^{\mathcal{CL}}$.

□

Lemma 8.8 $\mathsf{update}^{\mathcal{AL}} \propto \mathsf{update}^{\mathcal{CL}}$.

Proof Updating a liveness description only has an effect on the liveness component, in the concrete domain, as well as in the abstract domain. Therefore, it suffices to show that the concretisation of the abstract liveness component is always an approximation of the concrete liveness components for environments which are approximated by the abstract environment. Given Lemma 8.4 which proves that $\mathsf{extend}_a \propto \mathsf{extend}$, and the definition of live (Equation 8.1) and live_a (Equation 8.3), we can easily show that $\mathsf{live}_a \propto \mathsf{live}$, hence $\mathsf{update}^{\mathcal{AL}} \propto \mathsf{update}^{\mathcal{CL}}$.

□

We illustrate the derivation of liveness information with the deterministic version of append, c.f. Example 8.7.

Example 8.10 *Consider the code of* append *and the definition of the concrete description* CL_0 *from Example 8.7. Consider the abstract liveness call description* $AL_0 = \langle\{\left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right)\},\{Y^{\overline{e}}, X^{\overline{([],1)}}\},\{\,\}\rangle$. *Concrete calls covered by this description may at most have some sharing between the elements of* X *and* Y*, and only parts of* Y *may be live outside the call. We have* $CL_0 \sqsubseteq_{cl} \gamma^{\mathcal{L}}(AL_0)$.

Table 8.2 details the descriptions obtained at each of the program points.

Comparing this table with the table obtained for the concrete call CL_0 *from Example 8.7, we can draw the same conclusions as there: calling* append(X,Y,Z) *with its second list live, and the elements of* X *and* Y *aliased, leaves a dead list cell at the deconstruction at program point* (4).

The exit description as a result of analysing append/3 *under these conditions has only one relevant component, namely the structure sharing that the procedure call may generate.*

1	F_1	$=$	$\{Y\}$
	AL_1	$=$	AL_0
2	F_2	$=$	$\{\,\}$
	AL_2	$=$	$AL_0 = \begin{cases} \overline{A} & = & \{\left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right)\} \\ \overline{L} & = & \{Y^{\overline{\epsilon}}, X^{\overline{([],1)}}\} \end{cases}$
3	F_3	$=$	$\{Y, Xe, Xs\}$
	AL_3	$=$	$\begin{cases} \overline{A} & = & \{\left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right)\} \\ \overline{L} & = & \{Y^{\overline{\epsilon}}, X^{\overline{([],1)}}, Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}\} \end{cases}$
4	F_4	$=$	$\{Xe, Zs\}$
	AL_4	$=$	$\begin{cases} \overline{A} & = & \left\{ \begin{array}{l} \left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right), \left(X^{\overline{([],1)}} - Xe^{\overline{\epsilon}}\right), \\ \left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}}\right), \left(Xe^{\overline{\epsilon}} - Y^{\overline{([],1)}}\right) \end{array} \right\} \\ \overline{L} & = & \{Y^{\overline{\epsilon}}, Zs^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, X^{\overline{([],1)}}\} \end{cases}$
5	F_5	$=$	$\{\,\}$
	AL_5	$=$	$\begin{cases} \overline{A} & = & \left\{ \begin{array}{l} \left(X^{\overline{([],1)}} - Y^{\overline{([],1)}}\right), \left(X^{\overline{([],1)}} - Xe^{\overline{\epsilon}}\right), \\ \left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}}\right), \left(Xe^{\overline{\epsilon}} - Zs^{\overline{([],1)}}\right), \\ \left(Xs^{\overline{([],1)}} - Zs^{\overline{([],1)}}\right), \left(Y^{\overline{\epsilon}} - Zs^{\overline{\epsilon}}\right), \\ \left(X^{\overline{([],1)}} - Zs^{\overline{([],1)}}\right) \end{array} \right\} \\ \overline{L} & = & \{Y^{\overline{\epsilon}}, Zs^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, Xs^{\overline{([],1)}}, X^{\overline{([],1)}}\} \end{cases}$

Table 8.2: Abstract liveness details for the deterministic append procedure (Example 8.10). Forward use is listed as F_i, $1 \leq i \leq 5$, while the backward use sets are empty sets.

8.5 Increased Precision by Differential Semantics

In the previous sections, the concrete and abstract instantiations for the update function rely on extend, resp. extend_a:

$$\begin{array}{lcl} \mathsf{update}^{\mathcal{CL}}(i, CL_o, CL_n) & = & \left\{ \langle e, C, L\rangle \;\middle|\; \begin{array}{l} \langle e_o, C_o, L_o\rangle \in CL_o, \\ \langle e_n, C_n, L_n\rangle \in CL_n, \\ e_o \models e_n, C_o \subseteq_c C_n, \\ e = e_n, C = C_n, \\ L = \mathsf{live}(i, \langle e, C\rangle, L_o) \end{array} \right\} \\ \mathsf{update}^{\mathcal{AL}}(i, \langle \overline{A}_0, \overline{L}_0\rangle, \langle \overline{A}, \overline{L}\rangle) & = & \langle \overline{A}, \mathsf{live}_a(i, \overline{A}, \overline{L}_0)\rangle \end{array}$$

where

$$\begin{array}{lcl} \mathsf{live}(i, \langle e, C\rangle, L_0) & = & \langle e, \mathsf{extend}(L_0 \cup \mathsf{data}(U_i), C)\rangle \\ \mathsf{live}_a(i, \overline{A}, \overline{L}_0) & = & \mathsf{extend}_a(\overline{L}_0 \sqcup_{ad} \mathsf{data}_a(U_i), \overline{A}) \end{array}$$

where $U_i = \mathsf{forward}(i) \cup \mathsf{backward}(i)$.

Previously we have shown that extend is idempotent, while extend_a is not. This has consequences on the precision of the results for the abstract domain.

Example 8.11 *Consider a procedure p with abstract call description $\langle \overline{A}, \overline{L}_0\rangle$, where $\overline{A} = \{(\alpha - \beta), (\beta - \gamma)\}$ and $\overline{L}_0 = \{\alpha, \beta\}$. Example 8.9 showed that this call description is perfectly acceptable as in the abstract domain liveness information does not need to be synchronised with the structure sharing component.*

Let (i) be a program point in p, then the liveness component at program point (i) is obtained by extending the liveness component of the call description of the procedure to which (i) belongs, and the local in use information, with the structure sharing information at that program point. Let us assume that (i) is the first encountered program point in (p), this means that the structure sharing is still $\overline{A}$. If the in use information at (i) is the empty set, then then the liveness component computed at program point (i) is: $\overline{L}_i = \mathsf{extend}(\overline{L}_0, \overline{A}) = \{\alpha, \beta, \gamma\}$. This means that γ is now suddenly considered live, although in the call description it was not.

In the concrete domain, this overestimation does not occur as every concrete structure sharing described by $\overline{A}$ will either have α shared with β or β with γ but never both in the same liveness description. Hence, if α is live in the first case, then only β will be live. Given the fact that extend *is idempotent, extending a live set a number of times with the same set of structure sharing sets will always yield the same results.*

The imprecision arises from the fact that the liveness of the call description, which was already extended once with the initial structure sharing of that call, is extended w.r.t. to that initial structure sharing again at each program point in the procedure looked at.
In Chapter 6 we have shown that the concrete structure sharing at a program point can always be computed as the combination of the initial structure sharing of the call, and the local structure sharing that is built up by the literals in

the procedure. We can therefore reason that when computing a new liveness description, the liveness of the call description, *i.e.*, $\overline{L}_0$, needs only to be extended w.r.t. the local structure sharing (instead of the global structure sharing), while the in use information has to be extended w.r.t. the global structure sharing as usual. We redefine live using the initial structure sharing and local structure sharing as follows:

$$\begin{array}{l}\mathsf{live}(i, \langle e_0, C_0\rangle, \langle e_{l,i}, C_{l,i}\rangle, L_0) = \\ \quad \text{let } U_i = \mathsf{data}(\mathsf{forward}(i) \cup \mathsf{backward}(i)) \text{ in} \\ \quad \text{let } \langle e, C\rangle = \mathsf{comb}_c(\langle e_0, C_0\rangle, \langle e_{l,i}, C_{l,i}\rangle) \text{ in} \\ \quad\quad \langle e, \mathsf{extend}(L_0, C_{l,i}) \cup \mathsf{extend}(U_i, C)\rangle\end{array} \tag{8.5}$$

where $i \in \mathtt{pp}$, $e_0, e_{l,i} \in Eqn^+$, $C_0, C_{l,i} \in \wp(\mathcal{SD}_{VI})$, $L_0 \in \wp(\mathcal{D}_{VI})$, and where comb_c is the combination operator for concrete structure sharing tuples (Definition 6.24). In the abstract domain, we can then correctly approximate this operation using the following new definition of live_a:

$$\begin{array}{rcl}\mathsf{live}_a(i, \overline{A}_0, \overline{A}_{l,i}, \overline{L}_0) & = & \text{let } \overline{U}_i = \mathsf{data}_a(\mathsf{forward}(i) \cup \mathsf{backward}(i)) \text{ in} \\ & & \text{let } \overline{A} = \mathsf{comb}_a(\overline{A}_0, \overline{A}_{l,i}) \text{ in} \\ & & \quad \mathsf{extend}_a(\overline{L}_0, \overline{A}_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \overline{A})\end{array} \tag{8.6}$$

Given the fact that $\mathsf{extend}_a \propto \mathsf{extend}$, $\mathsf{comb}_a \propto \mathsf{comb}$ and obviously $\mathsf{data}_a \propto \mathsf{data}$, we can safely claim that $\mathsf{live}_a \propto \mathsf{live}$.

This definition of the liveness information clearly demands a semantics where the local information is separated from the global information. This can be achieved through the differential semantics. We therefore need to augment and instantiate $\mathcal{S}em_{M\delta}$ accordingly.

Let $\mathcal{S}em_{M\delta^+}$ be the augmented differential semantics with semantic functions $\mathbf{R}_{M\delta^+}$, $\mathbf{P}_{M\delta^+}$, $\mathbf{G}_{M\delta^+}$, etc. All these semantic functions are defined as in the differential semantics, except for the definition of literal goals and procedure calls.

$$\begin{array}{rcl}\mathbf{G}_{M\delta^+}[\![l]\!](e, A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } \mathcal{S}'_g = \mathsf{combupdate}(\mathtt{pp}(l), \mathcal{S}_g, \mathcal{S}_l) \text{ in} \\ & & \quad (\mathbf{L}_{M\delta^+}[\![l]\!]\, e\, \mathcal{S}'_g\mathcal{S}_l, \\ & & \quad\quad A[(\mathtt{pp}(l), \mathcal{S}_g), \mathcal{S}'_g]) \\ \mathbf{L}_{M\delta^+}[\![\mathsf{unif}]\!]\, e\, \mathcal{S}_g\mathcal{S}_l & = & \mathsf{add}(\mathsf{unif}, \mathcal{S}_l) \\ \mathbf{L}_{M\delta^+}[\![p(\overline{X})]\!]\, e\, \mathcal{S}_g\mathcal{S}_l & = & \mathsf{comb}(\mathcal{S}_l, e(p(\overline{X}), \mathcal{S}_g))\end{array}$$

In this definition, we use a new combupdate function that is a hybrid operation between the usual comb and update operations: it produces a new liveness description in which the structure sharing is the result of combining the structure sharing of the global and local liveness descriptions, and where the current liveness component is computed using the new live and live_a-functions described above. Instead of introducing a new auxiliary function combupdate, we could

have formulated the clauses using comb and update, yet this would require special attention as to the correct ordering (the update needs to be done using a separate global and local description), but also to a correct definition of comb (applying $\mathsf{comb}^{\mathcal{CL}}$ or $\mathsf{comb}^{\mathcal{AL}}$ (Definition 8.16, Definition 8.19 resp.) on the result of updating the liveness descriptions has the effect of reinitialising the liveness description). Therefore, a cleaner approach is to replace the normal functionality of update by a function that not only affects the liveness component, but also the structure sharing component. To make the change in functionality explicit, we replace the auxiliary operation update with a new operation called combupdate that makes these subtleties explicit.

We instantiate $\mathcal{S}em_{M\boldsymbol{\delta}^+}$ with $\mathcal{CL}$ as follows.

Definition 8.20 (Differential Concrete Liveness Derivation) *The differential concrete liveness derivation is defined as the augmented differential semantics $\mathcal{S}em_{M\boldsymbol{\delta}^+}$ instantiated with the domain $\mathcal{CL}$, where the auxiliary functions* init, comb, add, *are instantiated with the functions* $\mathsf{init}^{\mathcal{CL}}$, $\mathsf{comb}^{\mathcal{CL}}$, $\mathsf{add}^{\mathcal{CL}}$ *as defined in Definition 8.16, and where the new* combupdate *operation is instantiated with the function*

$$\begin{aligned}&\mathsf{combupdate}^{\mathcal{CL}}(i, CL_g, CL_l)\\ &= \left\{ \langle e, C, L\rangle \;\middle|\; \begin{array}{l} \langle e_g, C_g, L_g\rangle \in CL_g,\\ \langle e_l, C_l, L_l\rangle \in CL_l,\\ L = \mathsf{live}(i, \langle e_g, C_g\rangle, \langle e_l, C_l\rangle, L_g),\\ \langle e, C\rangle = \mathsf{comb}_c(\{\langle e_g, C_g\rangle\}, \{\langle e_l, C_l\rangle\}) \end{array} \right\}\end{aligned}$$

where live *is defined by Equation (8.5) and* comb_c *is given by Definition 6.24 (Page 120).*

As the structure sharing information in the context of the natural semantics $\mathcal{S}em_M$ is equivalent to the structure sharing information obtained in the differential semantics setting $\mathcal{S}em_{M\boldsymbol{\delta}}$, we may safely conclude that this remains true in the augmented semantics. Given that equivalence and the immediate dependence of liveness information on structure sharing, we can safely claim that $\mathcal{S}em_{M\boldsymbol{\delta}^+}(\mathcal{CL})$ is equivalent to $\mathcal{S}em_{M^+}(\mathcal{CL})$.

Definition 8.21 (Differential Abstract Liveness Derivation) *The differential abstract liveness derivation is the augmented differential semantics $\mathcal{S}em_{M\boldsymbol{\delta}^+}$ instantiated with the domain $\mathcal{AL}$, where the auxiliary functions* init, comb, add *are instantiated with the operations* $\mathsf{init}^{\mathcal{AL}}$, $\mathsf{comb}^{\mathcal{AL}}$ *and* $\mathsf{add}^{\mathcal{AL}}$ *resp., as defined in Definition 8.19, and where the new* combupdate *operation is instantiated with the function*

$$\begin{aligned}&\mathsf{combupdate}^{\mathcal{AL}}(i, \langle \overline{A}_g, \overline{L}_g\rangle, \langle \overline{A}_l, \overline{L}_l\rangle)\\ &= \langle \mathsf{comb}_a(\overline{A}_g, \overline{A}_l), \mathsf{live}_a(i, \overline{A}_g, \overline{A}_l, \overline{L}_0)\rangle\end{aligned}$$

where live_a *is defined by Equation (8.6).*

8.6 Related Work

Liveness information for logic programs was first detailed by Mulkers (1991). Roughly speaking the derived liveness information is similar to the liveness as derived here: it is also represented as a tuple consisting of a component representing the bindings of the variables (using the notion of a *concrete term environment*), a component representing their sharing, and finally, a component representing the actual live data structures. The differences lie mainly in the complexity of the underlying domains due to the absence of type and mode information in the underlying language (Prolog), and the actual formalisation of the derivation process which is done in terms of the well known generic framework for abstract interpretation for logic programs developed in (Bruynooghe 1991). Here, we give a simplified definition, adapted to the semantics of Mercury programs and formulated as an annotation of the source code with the clear aim of being used for a next analysis phase: reuse analysis. Formulating liveness analysis as we did also enables us to discuss more easily about the modularisation of that process, which is the aim of Chapter 10.

To our knowledge, the work of Mulkers (1991) and (Bruynooghe, Janssens, and Kågedal 1997) is the only other complete formalisation of liveness analysis expressed in the context of logic programming.

8.7 Conclusion

In this chapter we have given the definition of concrete and abstract liveness domains, and used these domains in the context of a number of different semantics. We first used the natural semantics $\mathcal{S}em_M$, and argued that these semantic functions needed to be adapted so as to be able to correctly derive liveness information. We called the result the *augmented natural semantics* for Mercury, and named it $\mathcal{S}em_{M^+}$. Yet, as we instantiated this semantics with the abstract liveness domain, we showed that due to the fact that the extension operation on which the propagation of liveness information is based, is not idempotent in the abstract domain, we obtain a possible loss of precision. This loss of precision can be alleviated by separating the global structure sharing information from the local part. This brought us to a new definition of obtaining the concrete liveness information at a program point. This new definition needs the separation of a local and global structure sharing, hence bringing us to the differential semantics. By augmenting that semantics, we obtained $\mathcal{S}em_{M\delta^+}$.

Figure 8.1 gives an overview of the different liveness derivations defined in this chapter.

While $\mathcal{S}em_{M^+}(\mathcal{CL})$ and $\mathcal{S}em_{M\delta^+}(\mathcal{CL})$ are equivalent, our previous discussion shows that $\mathcal{S}em_{M^+}(\mathcal{AL}) \propto \mathcal{S}em_{M\delta^+}(\mathcal{AL})$, and therefore $\mathcal{S}em_{M\delta^+}\mathcal{AL}$ is inherently more precise than $\mathcal{S}em_{M^+}(\mathcal{AL})$.

$$\begin{array}{ccc} \mathcal{S}em_{M^+}(\mathcal{CL}) & \Leftrightarrow & \mathcal{S}em_{M\delta^+}(\mathcal{CL}) \\ \text{(Definition 8.16)} & & \text{(Definition 8.20)} \\ \uparrow\propto & & \uparrow\propto \\ \mathcal{S}em_{M^+}(\mathcal{AL}) & \xrightarrow{\propto} & \mathcal{S}em_{M\delta^+}(\mathcal{AL}) \\ \text{(Definition 8.19)} & & \text{(Definition 8.21)} \end{array}$$

Figure 8.1: Overview of the instantiated liveness semantics for Mercury.

Chapter 9

Reuse Analysis, First Prototype Implementation

In this chapter we define the notions of data structure reuse, and report on our first prototype implementation of the liveness analysis presented in the previous chapter.

9.1 Structure Reuse, Terminology

The purpose of liveness analysis is to find opportunities for reusing heap cells as soon as they become *dead*. Indeed, liveness information may be interesting as such, but it is especially interesting to know the data structures that are *not* live. Given the safeness of our abstract domain in combination with the natural augmented semantics, we know that at each program point we obtain an over-estimation of the live heap cells. Therefore, if a specific data structure does not belong to the current liveness component of a program point, then the current literal definitely has a unique reference to that data structure, if at all.

Example 9.1 *Consider the procedure definition of the deterministic version of* append *given in Example 4.4 (page 42). Using the natural augmented semantics instantiated with the abstract domain of liveness descriptions,* i.e., $\mathcal{S}em_{M^+}(AL*)$*, we derive that for the abstract call description* $\langle\{\,\}, \{Z^{\overline{e}}\}\rangle$—i.e., *the input arguments do not share, and only the output argument* Z *is used and thus live in the calling environment of* append *— the liveness description at program point* (3) *consists of the tuple* $\langle\{\,\}, \{Y^{\overline{e}}, Xe^{\overline{e}}, Xs^{\overline{e}}, Z^{\overline{e}}\}\rangle$.

Indeed, before executing the literal at program point (3)*, no sharing exists between the available instantiated arguments* X *and* Y *(*$A_3 = \{\,\}$ *in Table 9.1). At* (3) *the variables* Y, Xe, Xs *are in forward use. As* append *is written in its deterministic version,*

pp	$F_{\mathtt{pp}}$	$A_{\mathtt{pp}}$	$L_{\mathtt{pp}}$
1	$\{Y\}$	$\{\,\}$	$\{Y^{\overline{\epsilon}}, Z^{\overline{\epsilon}}\}$
2	$\{\,\}$	$\{\,\}$	$\{Z^{\overline{\epsilon}}\}$
3	$\{Y, Xe, Xs\}$	$\{\,\}$	$\{Y^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}, Z^{\overline{\epsilon}}\}$
4	$\{Xe, Zs\}$	$\{\left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}}\right), \left(X^{\overline{([],1}} - Xe^{\overline{\epsilon}}\right)\}$	$\{Xe^{\overline{\epsilon}}, X^{\overline{([],1)}}, Zs^{\overline{\epsilon}}, Z^{\overline{\epsilon}}\}$
5	$\{\,\}$	$\left\{\begin{array}{l}\left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}}\right),\\ \left(X^{\overline{([],1}} - Xe^{\overline{\epsilon}}\right),\\ \left(Zs^{\overline{([],1)}} - X^{\overline{([],1)}}\right),\\ \left(Zs^{\overline{\epsilon}} - Y^{\overline{\epsilon}}\right)\end{array}\right\}$	$\{Z^{\overline{\epsilon}}\}$

Table 9.1: Liveness descriptions for the deterministic procedure append assuming the liveness call description $\langle\{\,\}, \{Z^{\overline{\epsilon}}\}\rangle$. $F_{\mathtt{pp}}$ represents the variables in local forward use (c.f. Example 7.3), $A_{\mathtt{pp}}$ is the goal-independent structure sharing information, while $L_{\mathtt{pp}}$ represents the liveness information for each program point.

no variables are in backward use. Hence, the set of live data structures at (3) *consists of* $\{Y^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, Xs^{\overline{\epsilon}}, Z^{\overline{\epsilon}}\}$*, where* $Z^{\overline{\epsilon}}$ *is due to the calling environment in which it is live.*

With this result we see that $X^{\overline{\epsilon}}$ *is not part of the current liveness information at that program point. This means that the deconstruction has a unique reference to the immediate heap cells to which* $X^{\overline{\epsilon}}$ *refers. If the deconstruction succeeds, then it creates two new references to the subterms of the main list-functor. While the structures pointed at by these two new references may still be needed during the rest of the program, we have the guarantee that the heap cells storing the main list cell have become garbage. Figure 9.1 gives a graphic representation of this situation.*

The liveness descriptions obtained for each of the program points in append *are shown in Table 9.1.*

In the previous example we specifically looked at the deconstruction literal for commenting on the heap cells that might become garbage. This is not without reason. Indeed, of all literal types, deconstruction unifications are the only literals where last and unique references to specific heap cells are released, and where we know the shape of what is released. The former is important because of course we want to detect as soon as possible where garbage cells are produced when executing the program, and the latter factor is of interest as we can then decide at compile-time what to do with these available heap cells.

We introduce the following terminology:

Definition 9.1 (Top Level Data Structure) *Consider a concrete data structure* $\langle e, X^s\rangle$*, then the heap cells used to store the outermost functor of the term* X^s *(in the context* e*),*

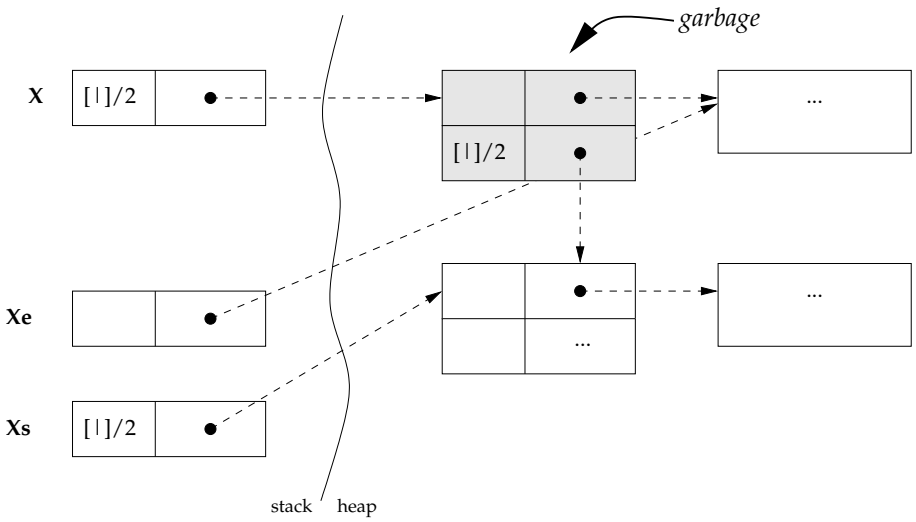


Figure 9.1: Memory reuse possibilities of the deterministic append procedure: $X^{\overline{e}}$ is not live, therefore the cells representing the main list functor (shown in grey) are definitely garbage after the deconstruction at program point (3) in Example 9.1.

is called the top level data structure *of X^s. We use the same notion for abstract data structures: the top level data structure for $X^{\overline{s}}$ consists of the heap cells used to store the outermost functors of the terms that may be bound to the subterms $X^{\overline{s}}$ of X.*

Usually, the notions of top level data structures are used in the context of deconstructions and constructions, in which case it is perfectly possible to determine the shape of the top level data structure of the deconstructed variable.

Example 9.2 *The top level data structure for the abstract value $X^{\overline{e}}$ in the context of Example 9.1 are the heap cells marked as garbage in Figure 9.1.*

Example 9.3 *Let X be bound to a term $f(g(1), h(2))$, then the heap cells needed to store the functor $f/2$ and its arguments* (i.e., *the references to its arguments) represent the top level data structure of X.*

Definition 9.2 (Available for Reuse) *The top level data structure of a data structure is said to be* available for reuse *iff these heap cells are not live.*

In the context of a specific deconstruction $X => f(Y_1, \ldots, Y_n)$ at a program point (i) with concrete liveness description CL (or abstract liveness description AL), the top level data structure of X^ϵ (or in the abstract domain $X^{\overline{e}}$) becomes available for reuse *in CL (resp. AL) iff X^ϵ (resp. $X^{\overline{e}}$) is not live according to the current liveness component in CL (resp. AL).*

Example 9.4 *In Example 9.1, the top level data structure of* $X^{\overline{\epsilon}}$ *become available for reuse at program point* (3).

Just as the name suggests, top level data structures that are available for reuse can be used for allocating new terms on the heap. This reallocation can be decided at compile-time given the fact that we know the shape and therefore the size of the available heap cells. The simplest form of reallocation is the situation where a deconstruction is followed by a suitable construction unification both covered by the same execution path.

Definition 9.3 (Matching Deconstruction-Construction Pair) *We say that a deconstruction literal* $X => f(Y_1, \ldots, Y_n)$ *and a construction literal* $Z <= g(T_1, \ldots, T_m)$ *are a* matching *pair iff*

- *there exists an execution path*[1] *to which both literals belong, and the deconstruction occurs before the construction*[2]*,*
- *the size of the heap cells used for the top level data structure of the deconstructed data structure is at least as big as the size of the heap cells used for the top level data structure of the constructed data structure.*

A matching deconstruction-construction pair is called a *perfect* match if the top level data structures of the deconstructed and constructed data structures represent the same functor. A matching pair is called a *near* match if the involved top level data structures represent functors with the same arity.

All matching deconstruction-construction pairs of a program are a potential source for reusing heap cells. Yet some of these matching pairs may overlap when they share the same deconstruction. We introduce the notion of a candidate for reuse:

Definition 9.4 (Candidate for Reuse) *In a matching deconstruction-construction pair, the term that is constructed in the construction literal is a* candidate for reuse *for the term that is deconstructed in the deconstruction literal.*

One deconstruction can have multiple candidates for reuse.

We define the notions of *direct reuse* and *indirect reuse*.

Definition 9.5 (Direct Reuse) *Given a procedure with call description* $AL \in \mathcal{AL}$*, then this procedure is said to have opportunities of* direct reuse *if it contains matching deconstruction-construction pairs in which the top level data structures of the deconstructed data structures become available for reuse for that call description.*

[1] Hence, both literals belong to the same procedure definition.

[2] Recall that we assume that the analysis passes when implemented in a compiler, may not be followed by any compiler pass that would possibly reorder the literals in a procedure definition. This means that all subsequent passes of the compiler must be *order preserving* w.r.t. the literals.

The notion of indirect reuse applied to procedures is defined in a recursive way:

Definition 9.6 (Indirect Reuse) *A procedure with call description $AL \in \mathcal{AL}$ is said to have opportunities of* indirect reuse *if it contains calls to procedures with direct or indirect reuse.*

To distinguish the phase of merely deriving liveness analysis from the phase in which we check for direct or indirect reuses, we call the latter *reuse analysis*.

Definition 9.7 (Reuse Analysis) *Reuse analysis is the process of checking for direct and indirect reuses in called procedures.*

In the previous chapter we have defined the liveness annotation process using the goal-dependent natural semantics of Mercury programs. This yields a new set of program point annotations for each encountered call description. Hence, the liveness analysis is polyvariant. For each encountered call description, reuse analysis may discover a different set of reuse opportunities. If these reuse opportunities were to be compiled into low-level program code realising the actual structure reuse, then for each call description, a new *version* for the procedure should be generated. Yet in general, we do not distinguish versions of a procedure by the call description with which they are called, but by the combinations of the reuse opportunities they allow. We therefore use the following definition:

Definition 9.8 (Reuse Version of a Procedure) *A procedure may be decoupled in a number of versions. Each version repeats the program code of the original procedure definition, yet explicitly annotates or implements the possibilities of direct reuse, and explicitly implements indirect reuse by calling the adequate reuse version of the called procedure.*

Note that in the previous definition, we do not expect that each version actually implements the direct reuses that were detected by the reuse analysis as the real reuse can only be possible once the results of the analyses are truly used by a Mercury compiler.

In the remainder of this chapter we describe our first prototype implementation for detecting direct and indirect reuse. The purpose of this prototype was to evaluate the usefulness and feasibility of liveness analysis in order to motivate a more complete integration of a real liveness and reuse system in the Melbourne Mercury compiler.

9.2 Prototype Description

Our first prototype system was a system mainly written in Prolog. This system is based on the "Abstract Machine for Abstract Interpretation", in short AMAI (Jans-

sens, Bruynooghe, and Dumortier 1995)—a generic tool for the abstract interpretation of untyped logic programming languages without declarations. This engine had to be extended in a number of ways:

- The AMAI is meant to deal with pure conjunctive goals. Disjunctions were unknown. Although it is easy to replace explicit disjunctions in Mercury source code by a call to a new predicate that is defined by a number of clauses, each clause corresponding to one branch of the disjunction, we chose to adapt the AMAI instead:
  - Mercury makes extensive use of disjunctions (as all clauses defining a predicate are always replaced by one single disjunctive goal), therefore such transformation would change our source code too much;
  - in this setting we are searching for matching deconstruction/construction pairs, *i.e.*, reuse takes only place within the same procedure definition. By replacing disjunctions by predicate calls we loose a considerable number of reuse opportunities w.r.t. the original source code.

  Note that we did not specifically extend the AMAI to be able to cope with if-then-else constructs, as they can be transformed into disjunctions without loss of precision in the obtained results.
- The AMAI is a tool for analysing untyped logic languages with neither mode nor determinism information. We added some extensions to cope with this extra information that is available in Mercury programs;

Input to the analysis engine is a set of Prolog-facts representing Mercury code as defined in Chapter 4. We call these facts AMAI *instructions*. Initially, these instructions had to be produced manually, yet soon we added a hook to the Melbourne Mercury compiler[3] that produced these facts from real Mercury programs automatically. This hook intercepts the normal compilation, and uses the high level internal representation[4] of the initial program to produce the required Prolog-facts. These instructions also contain the necessary information about types, modes and determinism. Hence we start with a situation where all required implicit information as described in Section 4.2 is present.

The output of the prototype system is basically a set of annotation tables. No compiled code taking advantage of the discovered reuse opportunities is generated. For ease of inspection, we also generate an HTML-file (W3C) that gives a view on each of the reuse versions of the encountered procedures, highlighting all the reuse possibilities.

Figure 9.2 gives an overview of the structure of the prototype system.

[3] At that moment we used version 0.8 of that compiler.

[4] This high level internal representation is commonly called the "High Level Data Structure", or HLDS in short.

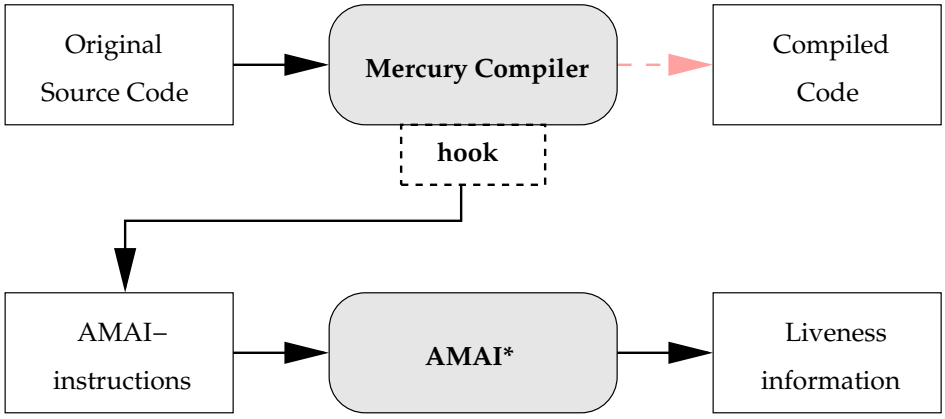


Figure 9.2: Structure of the prototype liveness analysis system developed for Mercury.

In the remainder of this section we sketch some of the design decisions for each of the annotations and analyses involved.

9.2.1 Forward Use, Backward Use

Forward use information is computed as described in Section 7.2. Backward use information is derived using a specific backward use analysis as described in section 7.4.

The programs are pre-annotated with forward use as well as backward use before starting the liveness analysis.

9.2.2 Abstract Liveness Descriptions

In this prototype we use the abstract domain $\mathcal{AL}$. Recall that elements in this domain consist of tuples containing only two components: a structure sharing component and a liveness component.

9.2.2.1 Structure Sharing

We represent structure sharing as unordered lists of unordered pairs of data structures. A data structure is represented as a variable adorned with a selector. A selector is a list of unit selectors. A unit selector consists of a description of a functor (*i.e.*, its name and arity), and a natural number n selecting the n^{th} child of the subterm with that functor.

9.2.2.2 Liveness Component

The liveness component is represented by an unordered list of data structures. Each data structure is represented as in the structure sharing component.

9.2.3 Liveness Analysis

The analysis system follows the natural augmented semantics $\mathcal{S}em_{M^+}$. In Section 8.5 we showed that a differential view on the structure sharing may improve the precision of the obtained results, yet this was not implemented at that stage.

It implements $\mathcal{S}em_{M^+}$ by a top-down analysis. The result of the analysis is a table containing the call-exit descriptions of the encountered procedures, and an annotation table annotating the individual program points with the liveness descriptions per call description of a procedure. The analysis is polyvariant: for each call description, a different series of program point annotations is obtained. Per call description we associate a new *version* of the procedure considered.

The prototype does not deal with modules, hence considers programs to be one single monolithic block.

9.2.4 Reuse Analysis

Using the annotations, we perform a bottom-up check of the reuse opportunities for each call description for each procedure. This process checks for direct reuse in the called procedures. Indirect reuse follows automatically if at some program point the called procedure allows direct or indirect reuse. We only allow perfect matching deconstruction-construction pairs. If a deconstruction has multiple reuse candidates in the same execution path, then we select the construction literal that follows the deconstruction the closest.

By the end of this analysis we obtain direct and indirect reuse annotations for each of the versions derived by the liveness analysis.

9.3 Benchmark: `labelopt`

The prototype system was first tested on some of the classical benchmarks such as list concatenation, naive reverse, or other typical list manipulations. Yet our main benchmark was the study of real life code, *i.e.*, one specific module of the Melbourne Mercury compiler, known for its high memory requirements: `labelopt`. The goal of this study was to discover how much structure reuse our prototype would be able to detect for some of the common call descriptions of the main procedures defined in that module.

We first detail the structure of this module. Section 9.3.2 gives a report on the opportunities of reuse discovered by our prototype system.

9.3.1 Code Structure and Potential Reuses

The analysed program, `labelopt`, is a module from the Mercury compiler. The main procedure exported by this module is labelopt_main:

```
:- pred labelopt_main(list(instruction), bool,
                      list(instruction), bool).
:- mode labelopt_main(in, in, out, out) is det.
```

The purpose of this procedure is to transform a list of program instructions into a new list of optimised instructions. The module uses procedures from two other modules, `opt_util` and `list`[5]. In a first test, we limit our analysis by substituting the few predicates from `opt_util` by dummy predicates, such that the liveness results for the main procedures within `labelopt` are not altered. The definitions of the procedures in `list` are kept, as these present a number of interesting opportunities for reuse.

Figures 9.3 and 9.4 show the (simplified) call graph of the main procedure labelopt_main, leaving out dummy calls. Procedures whose definitions contain matching deconstruction/construction pairs are marked with *D/C*. Recursive calls are indicated by loops in the call graph. The *D/C* annotations do not necessarily mean that structure reuse is definitely possible, they only indicate that the specific predicate may have some potential for direct reuse. A brief description follows.

- labelopt_main: This procedure transforms a list of instructions (the input) into a new list of instructions (the output). There are no matching deconstruction/construction pairs. Therefore possibilities of reuse need to be found down in the call graph.
- build_useset: Here a new set of labels is produced, starting from a list of instructions and an initially empty set of labels. Again, there are no matching deconstruction/construction pairs in the procedure definition.
- instr \ _list : In this procedure, a list of instructions is decomposed into its elements. A new list is generated either by explicitly taking over the elements of the input list, or by calling eliminate on that element, and incorporating the resulting list into the new list. The procedure is recursive, and contains matching deconstruction/construction pairs. Hence, it has opportunities for direct reuse.
- eliminate: Basically a variable of type pair, representing pairs of elements, is deconstructed and a new pair is created. The output of the procedure is a list containing that new pair. The procedure contains a matching deconstruction/construction pair, hence this procedure may be a candidate for direct reuse.

[5] In fact it uses also the `set`-module, but given the fact that a set is represented internally as a list, all procedures manipulating sets are simply calling list-procedures.

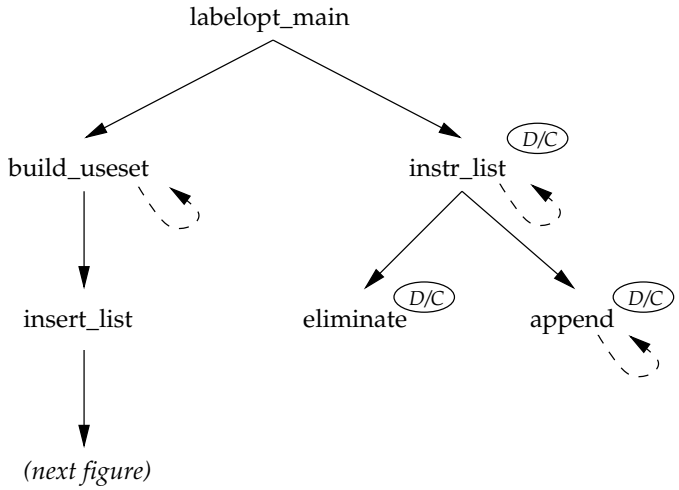


Figure 9.3: Call graph of labelopt_main. *D/C* indicates the presence of matching deconstruction/construction pairs. A loop represents recursive calls.

- list manipulation predicates: The remaining procedures are procedures that manipulate lists. There meaning should be to some extent obvious: append is the classical list-concatenation operation, insert_list inserts a list of elements into an existing set (which is represented as an ordered list). Procedure sort_and_remove_dups sorts a lists, and removes any duplicates. This is done by using merge_sort to order the list, and remove_adjacent_dups to remove the duplicates that are now always adjacent elements. Procedure merge_sort is written in terms of length — a procedure that returns the length of a list, split_list — a procedure to split a list in two, and merge — a procedure to merge two lists back together. Finally, merge_and_remove_dups merges two ordered lists, and removes the duplicates. Some of these procedures contain matching deconstruction/construction pairs, hence are candidates for direct reuse. Given the fact that most of these procedures are also recursive, the direct reuses should also achieve indirect reuses in these recursive calls.

Judging on the presence of matching deconstruction/construction pairs, the following procedures show potential for direct reuse: instr_list , eliminate of the specific labelopt procedures and from the list manipulating procedures: append, merge_sort, split_list , merge, remove_adjacent_dups and merge_and_remove_dups. Figures 9.5 and 9.6 present the relevant pieces of the definitions of these procedures.

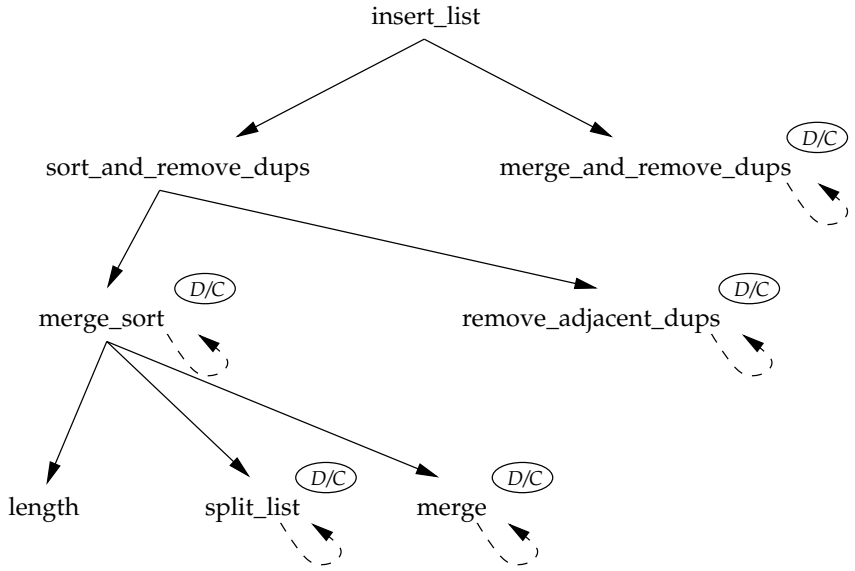


Figure 9.4: Call graph of insert_list /3. All predicates belong to the list-module. *D/C* indicates the presence of matching deconstruction/construction pairs. A loop represents recursive calls.

9.3.2 Identified reuses

Our analysis engine detects reuse of the top level data structures of the relevant variables in the procedures:

- instr_list ,
- eliminate,
- append,
- remove_adjacent_dups,
- and split_list .

Within each of these procedures a variable L of type list(T), is deconstructed in a liveness environment that does not comprise $L^{\overline{\epsilon}}$. For all these procedures direct reuse is detected, but also the recursive calls allow reuse of the top level data structure of the deconstructed lists. Thus, the complete input list can be reused if the source code is compiled to optimised code.

```
:- pred instr_list(list(instruction),
                   list(instruction)).
:- mode instr_list(in,out) is det.
instr_list([],[]).
instr_list(L1,L2):-
   (
      L1 == [], L2 := []
   ;
      L1 => [I0 | MoreL],
      I0 => Uinstr - _Comment,
      if (...) % perform some tests
      then (R <= [I0])
      else (
          eliminate(I0,R),
          opt_util_predicate(Uinstr,...)
      ),
      instr_list(MoreL,MoreR),
      append(R,MoreR,L2)
   ).
```

```
:- pred eliminate(instruction,
                  list(instruction)).
:- mode eliminate(in,out) is det.
eliminate(I,List):-
   I => In0 - C0,
   if (...) % perform some tests
   then (List <= [])
   else (
       In <= ... , % a constant
       NewI <= In - C0,
       List <= [NewI]
   ).
```

Figure 9.5: Relevant code of `labelopt`-predicates.

9.3.3 Undetected Possibilities of Direct Reuse

Although our analysis does detect many direct reuses, it does not detect them all. In the procedures where possible reuses remain undetected we see that the reason for these *misses* often lies in the fact that one single deconstruction is used to serve different purposes: it is a test to see whether a variable is bound to the desired outermost functor, it is used to select different arguments at the same time, although not all of them are always needed, etc. We discuss this using merge_sort and merge, c.f. Figure 9.6, two procedures in which this phenomenon occurs.

- merge_sort: The input list $L1$ is live in the program point of the relevant deconstruction literal as it is still needed in the second branch of the switch following that literal; however, if we check the liveness situation at the program point of the construction literal $L2 <= [E]$, we see that $L1^{\epsilon}$ is not live. So in a sense, we lost an opportunity for reuse as our reuse detection scheme does not perform the check at the construction. This problem can be solved by repeating the deconstruction. Indeed, a closer look at the procedure reveals that the literal $L1 => [E|R]$ has a double purpose: select the tail of the list needed for performing the switch; and select the first element in the same occasion which is only needed in the first branch of the switch. By decoupling the two goals of the deconstruction and therefore repeating the deconstruction just before literal $L2 <= [E]$ we could have automatically detected direct reuse of the involved top level data structure.

- merge: In merge and the very similar merge_and_remove_dups, no direct reuse is detected. We restrict the discussion to merge. The call description for merge(A,B,C) guarantees that there is no structure sharing between the input lists, and that neither the lists A nor B are initially live (*i.e.*, needed in the callers' context). Closer inspection of the code reveals that depending

```
:- pred append(list(T),list(T),list(T)).
:- mode append(in,in,out) is det.
append(X,Y,Z):-
    (
        X == [], Z := Y
    ;
        X => [X1|Xs],
        append(Xs,Y,Zs),
        Z <= [X1|Zs]
    ).

:- pred merge_sort(list(T),list(T)).
:- mode merge_sort(in,out) is det.
merge_sort(L1,L2):-
    (
        L1 == [], L2 <= []
    ;
        L1 => [ E | R ],
        (   % switch on R
            R == []
            L2 <= [ E ]
        ;
            R => [_|_],
            length(L1,Length),
            HL is Length // 2,
            (
                split_list(HL,L1,F,B),
                merge_sort(F,SF),
                merge_sort(B,SB),
                merge(SF,SB,L2)
            ;
                ... % error
            )
        )
    ).

:- pred split_list(int,list(T),
                   list(T),list(T)).
:- mode split_list(in,in,out,out)
                        is semidet.
split_list(N, List, Start, End):-
    if (N == 0)
    then (
        Start <= [],
        End := List
    ) else (
        N1 is N - 1,
        List => [Head | List1],
        split_list(N1, List1, Start1, End),
        Start <= [Head | Start1]
    ).
```

```
:- pred merge(list(T),list(T),list(T)).
:- mode merge(in,in,out) is det.
merge(A, B, C):-
    if (A => [X|Xs])
    then (
        if (B => [Y|Ys])
        then (
            if (compare(<, X, Y))
            then (
                Z := X,
                list_merge(Xs, B, Zs)
            ) else (
                Z := Y,
                list_merge(A, Ys, Zs)
            ),
            C <= [Z|Zs]
        ) else (C := A)
    ) else (C := B).

:- pred remove_adjacent_dups(list(T),
                             T,list(T)).
:- mode remove_adjacent_dups(in,in,out)
                                is det.
remove_adjacent_dups(L1,X,L2):-
    ( % switch on L1
        L1 == [], L2 <= [X]
    ;
        L1 => [ X1 | R ],
        if (X == X1)
        then (
            remove_adjacent_dups(R,X,L2)
        ) else (
            remove_adjacent_dups(R,X1,NR),
            L2 <= [ X | NR ]
        )
    ).

:- pred merge_and_remove_dups(list(T),
                              list(T),list(T)).
:- mode merge_and_remove_dups(in,in,out)
                                 is det.
merge_and_remove_dups(A, B, C):-
    % code very similar to merge/3
    merge(A,B,C).
```

Figure 9.6: Relevant code of list(T)-manipulating predicates.

on the situation, either the top level data structure of $A^{\overline{e}}$ or the top level data structure of $B^{\overline{e}}$ can safely be reused for the construction of list C. The reason why this remains undetected is that $A^{\overline{e}}$ is live at the deconstruction of A because depending on the value of B, A may be assigned to C. We have a similar situation for B. A closer inspection of the code again reveals that the deconstructions serve multiple purposes: check whether the lists have any elements at all, select the first elements (needed for comparison), and select the tails (needed for the recursive calls). By decoupling these different uses and again repeating the deconstructions, direct reuse could have been detected. The resulting code would be as in Figure 9.7.

The above examples are clearly examples in which direct reuse is not detected using the strategy we use in our analysis. We have two options. Either we

```
:- pred merge(list(T),list(T),list(T)).
:- mode merge(in,in,out) is det.
merge(A, B, C) :-
    if (A => [X|_])
    then (
        if (B => [Y|_])
        then (
            if (compare(<, X, Y))
            then (
                A => [_|Xs], % repeated!
                Z := X,
                list_merge(Xs, B, Zs)
            ) else (
                B => [_|Ys], % repeated!
                Z := Y,
                list_merge(A, Ys, Zs)
            ),
            C <= [Z|Zs]
        ) else (C := A)
    ) else (C := B).
```

Figure 9.7: Code for merge optimised for structure reuse.

try to adopt our strategy, which may clearly make the analysis more complex as it will have to cover more cases of direct reuse possibilities. Or, and this is our preferred option, we make sure that the output of our analysis is verbose enough so as to give enough hints to the interested programmer why some pieces of program code do not yield the intended optimisation. In the latter case, input of the programmer may be valuable in the sense that the programmer, eager to obtain optimised code w.r.t. memory usage, could annotate her/his program with pragmas making clear that she/he thinks that some reuse could be possible at the annotated program points.

Note that even by separating the global structure sharing from the local structure, hence computing liveness as described by Equation 8.6 (Page 185) would not have helped to detect these cases of direct reuse.

9.4 Prototype Evaluation

Table 9.2 presents the analysis results of the procedures as they appear in our experiment. The full code listing is given in Appendix A.

Each procedure is listed with a call description and information about the

Predicate	**Calldescr.** L_0	**Reuse**
`labelopt_main`(H_1, H_2, H_3, H_4)	$\{H_3^{\overline{\epsilon}}, H_4^{\overline{\epsilon}}\}$	i
`build_useset`(H_1, H_2)	$\{H_1^{\overline{\epsilon}}, H_2^{\overline{\epsilon}}\}$	i
`build_useset_2`(H_1, H_2, H_3)	$\{H_1^{\overline{\epsilon}}, H_3^{\overline{\epsilon}}\}$	i
`instr_list`$(H_1, H_2, H_3, H_4, H_5)$	$\{H_4^{\overline{\epsilon}}, H_5^{\overline{\epsilon}}\}$	$d+i$
`eliminate`(H_1, H_2, H_3, H_4)	$\{H_2^{\overline{\epsilon}}, H_3^{\overline{\epsilon}}, H_4^{\overline{\epsilon}}\}$	d
`eliminate`(H_1, H_2, H_3, H_4)	$\{H_1^{\overline{(-,1)}}, H_3^{\overline{\epsilon}}, H_4^{\overline{\epsilon}}\}$	d
`set_init`(H_1)	$\{H_1^{\overline{\epsilon}}\}$	no
`set_insert_list`(H_1, H_2, H_3)	$\{H_3^{\overline{\epsilon}}\}$	i
`set_member`(H_1, H_2)	$\{H_1^{\overline{\epsilon}}, H_2^{\overline{\epsilon}}\}$	no
`sort_and_remove_dups`(H_1, H_2)	$\{H_2^{\overline{\epsilon}}\}$	i
`merge_and_remove_dups`(H_1, H_2, H_3)	$\{H_3^{\overline{\epsilon}}\}$	no*
`member`(H_1, H_2)	$\{H_1^{\overline{\epsilon}}, H_2^{\overline{\epsilon}}\}$	no
`append`(H_1, H_2, H_3)	$\{H_3^{\overline{\epsilon}}\}$	$d+i$
`merge`(H_1, H_2, H_3)	$\{H_3^{\overline{\epsilon}}\}$	no*
`merge_sort`(H_1, H_2)	$\{H_2^{\overline{\epsilon}}\}$	no*
`length`(H_1, H_2)	$\{H_1^{\overline{\epsilon}}, H_2^{\overline{\epsilon}}\}$	no
`length_2`(H_1, H_2, H_3)	$\{H_1^{\overline{\epsilon}}, H_3^{\overline{\epsilon}}\}$	no
`split_list`(H_1, H_2, H_3, H_4)	$\{H_3^{\overline{\epsilon}}, H_4^{\overline{\epsilon}}\}$	d
`remove_adjacent_dups`(H_1, H_2)	$\{H_2^{\overline{\epsilon}}\}$	i
`remove_adjacent_dups_2`(H_1, H_2, H_3)	$\{H_3^{\overline{\epsilon}}\}$	d

Table 9.2: Overview of the analysed predicates in module `labelopt`. All call descriptions have an empty sharing component. d = direct reuse, i = indirect reuse, $d+i$ = combined reuse, no = no reuse possible, no* = possibly missed reuse.

reuse possibilities of the procedure called under that call situation. The reuse information states whether the procedure has possibilities for direct reuse (d), indirect reuse (i), or both direct and indirect reuse ($d+i$). If the procedure has no reuse at all, then we label it with "no". And finally, if the procedure contains matching deconstruction/construction pairs, or a call to a procedure for which a call description exists under which reuse is possible, yet no reuse is allowed in that procedure for that specific call, then we label the procedure call with no*.

A call description in $\mathcal{AL}$ contains a sharing component and a liveness component. In these examples, the sharing component happens to be always the empty set, hence we omit it in the table below and represent call descriptions by their liveness component only.

Our experiments were done on an UltraSPARC-IIi (333Mhz) with 256MB RAM, using SunOS Release 5.7, under a usual (small) workload. The Prolog-engine used was Master Prolog, release 4.1 ERP. On this platform the analysis of procedure labelopt_main in our module `labelopt` (already converted into AMAI instructions), took in average 2.84 seconds.

In a second experiment we explicitly included all predicates of the imported module `opt_util` instead of using dummy predicates. The resulting module herewith contained about 100 predicates (which is a rare situation for normal real-life projects using modules). The analysis of procedure labelopt_main under these conditions took more than 20 minutes. One reason for this high analysis time is to be found in the proliferation of the versions of the predicates (e.g. the analysis detects 7 different call descriptions for append/3, which means that append/3 is analysed 7 times — in three of the cases reuse is detected. This proliferation can be reduced by using call independent analysis results instead of having to reanalyse each procedure with its new call description. A second reason is related to the number of functors defining one single type. In our experiment, the concerned predicates mainly manipulate lists of instructions. The type instruction defines 26 different functors. When starting to select substructures below those functors, an explosive growth of the number of structure sharing possibilities occurs. This phenomenon is discussed in Section 11.3 where a widening operation is introduced to deal with such explosive structure sharing behaviour.

9.5 Conclusion

In this chapter we defined some new terminology to be able to discuss the reuses that can be derived starting from liveness information. We also described our very first and therefore basic prototype implementation of liveness analysis. This analysis yields very positive results for a real life module taken from the Melbourne Mercury compiler: the analysis has an acceptable performance, and detects a great number of possible reuses. The main drawback of the analysis is that it only handles code as one single monolithic block. Therefore the most needed extension at this stage to the liveness analysis and therefore its theory is to be able to cope with modules. To overcome this limitation, it becomes essential to develop a call independent approach.

The description of this prototype, as well as the evaluation of the results obtained with it are published in (Mazur, Janssens, and Bruynooghe 1999a).

Chapter 10

Module-enabled Structure Reuse Analysis

In the previous chapter we presented our first prototype implementation for deriving data structure reuse annotations of the form *direct reuse* and *indirect reuse*. The analysis performed well and yielded satisfying results, yet shows one main disadvantage: it can only handle programs as one single monolithic block.

In this chapter we present the underlying theory of how to enable liveness and reuse analysis to correctly deal with modules, yet preserving the potential of reuse available in a program.

We adapted our initial prototype and report on the obtained results in this chapter. The description of the actual integration of the analysis into the Melbourne Mercury compiler is postponed to Chapter 12 as we need some additional technical optimisations for better performance (Chapter 11).

10.1 Introduction

In Chapter 9 we separated deriving liveness descriptions from deriving the actual reuse opportunities. Thus, to enable the use of modules, we need to modularise both derivation systems.

Although liveness descriptions inherently have a goal-dependent nature due to the global structure sharing and global liveness information needed to correctly compute the live data structures at a specific program point within a procedure, we know that the structure sharing on which it depends can be derived in a goal-independent way. (Section 6.3.4, Theorem 6.1). We also know that the liveness derivation process is mainly structure sharing driven as the liveness component of the analysis domains (whether concrete or abstract) is not actually in-

volved in the fixpoint computation. This means that for the analysis of a given procedure with a given call description, it suffices to know the local structure sharing sets of each of the called procedures from within that procedure in order to derive correct liveness descriptions in that procedure. Thus we can expect that the modularisation of the liveness process can easily be achieved by replacing the differential view on the liveness derivation process by a goal-independent based view.

The modularisation of the reuse analysis part is more troublesome. In the Melbourne Mercury Compiler, all modules are analysed separately, and usually in a bottom up way. This means that if a module depends on another module (by the use of procedures defined in the latter), then the latter module is compiled before the former. In the context of our reuse analysis, this means that we need to compile procedures of which we don't know how they will be used.

We formulate both modularisation processes using the following example: consider a module m_1 and a module m_2. Module m_1 defines a procedure p_1 and module m_2 defines a procedure p_2. The procedure definition of p_1 contains a call to procedure p_2:

```
% module m1
% :- pred p1(...).
% :- mode p1(...).
p1(...) :-
    ... ,
    (i) p2(...), ...,
    (j) X => f(...), ...,
    (k) Y <= f(...), ... .

% module m2
% :- pred p2(...).
% :- mode p2(...).
p2(...) :- ... .
```

1. p_1 contains a matching deconstruction/construction pair. We want to verify whether or not direct reuse is possible given a specific liveness call description. In this case verification is reduced to computing the liveness description at the deconstruction literal of the matching pair, and verifying whether the top level data structure of the deconstructed variable is definitely dead or not. In order to compute that liveness component we need: forward use — a simple syntactical property, backward use — can also be reduced to a simple syntactical property, and structure sharing information. To be able to correctly determine the structure sharing information, we also need the structure sharing that can possibly be created by the call to p_2. This means that to determine the liveness descriptions in p_1 we only need the correct structure sharing information of p_2. This information can be provided by

a goal-independent analysis of the procedures defined in m_2. It suffices to record the results of such an analysis in a special optimisation interface file such that the information can be used for the compilation of modules depending on m_2, in our case module m_1. This problem is therefore straightforward. We will only briefly sketch its formal fundamentals.

2. The second issue is not so straightforward. Suppose p_2 might have some potential for reuses if called in the appropriate way, then how do we detect this without knowing how p_2 might be called from other modules? How do we decide to generate a specialised version for p_2? And finally, even if we generate an optimised version, how do we know when it is safe to call that version without compromising the safe execution of the program?

The first issue can simply be solved by recognising which information about a module is necessary in order to obtain correct liveness descriptions for the calls to procedures defined in module depending on it. In this case this simply consists of structure sharing information. The second issue is about deciding how to optimise procedures without knowing how these procedures are called, and what information to keep about these optimised procedures to guarantee their safe usage.

In the following sections we assume that the modules are not *mutually recursive* in the sense that procedures defined in a module m_1 can only call procedures defined in a module m_2 if m_2 does not contain any procedures that call a procedure defined in m_1. The reason for this restriction is that situations of mutually recursive modules are known to be harder to handle, especially in the context where normally each module is only compiled once. As a matter of fact, in order to achieve the same degree of precision for programs with mutually dependent modules w.r.t. programs in which the modules form a loop-free dependency tree, the fixpoint computation of the analysis processes must be defined over the boundaries of the modules, hence each module may have to be analysed several times. Writing programs with mutual dependent modules is usually seen as a bad programming habit. We therefore assume that in our theoretical setting modules are not mutually recursive. In practice, such programs could either undergo a special program transformation step that reorganises the modules accordingly, or, the compilation system could be adapted in such a way that program analyses have to reach a fixpoint over the boundaries of single modules (Bueno, García de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2001; Puebla, Correas, Hermenegildo, Bueno, García de la Banda, Marriott, and Stuckey 2004). See also Section 11.2, more specifically, page 251.

10.2 Modular Liveness Analysis

The issue of deriving liveness descriptions in the presence of modules can be solved by separating the liveness information from the structure sharing information, and deriving the latter by a goal-independent analysis. The main reason why we can separate liveness information from the structure sharing derivation process is the fact that liveness information does not participate in any fixpoint computation. No analysis as such is needed once all underlying information such as forward use, backward use and structure sharing information is known. Forward and backward use are both local properties, so in order to perform a (goal-dependent) liveness analysis of a procedure defined in a given module, it suffices to know the structure sharing that might be built by each of the called procedures. If these procedures are defined in separate modules, then these modules need to have been analysed w.r.t. structure sharing prior to the liveness analysis of the procedure of interest.

Formally, we replace the (augmented) differential abstract liveness derivation, represented by $\mathcal{S}em_{M\delta^+}(\mathcal{AL})$, by an augmented goal-independent based one, *i.e.*, $\mathcal{S}em_{M\bullet^+}(\mathcal{AL})$. We know that the pure differential semantics $\mathcal{S}em_{M\delta}$ is equivalent to the goal-independent based one, *i.e.*, $\mathcal{S}em_{M\bullet}$ (Theorem 5.6), therefore, if we show that the slight adaptations augmenting both semantics preserves this property, then using $\mathcal{S}em_{M\bullet^+}(\mathcal{AL})$ instead of $\mathcal{S}em_{M\delta^+}(\mathcal{AL})$ guarantees the correctness of the abstract instantiation.

In Chapter 5 we presented the goal-independent based semantics of a program as a semantics consisting of two parts: a pure goal-independent semantics, and a goal-dependent semantics based on that goal-independent semantics. When instantiating this semantics, both parts are instantiated by the same domain. When using the (augmented) goal-independent based semantics for the abstract liveness descriptions domain, it does not make much sense to instantiate the goal-independent part with that same domain. Indeed, we know that liveness information is a simple annotation depending on different underlying information, but it does not, in any sense, depend on the liveness information of the exit descriptions of analysed procedures. Therefore we can drop the liveness component from the domain when instantiating the goal-independent part of the semantics. Obviously, without liveness information, that goal-independent part need not to be augmented.

We obtain the following setting: we define the augmented goal-dependent liveness semantics $\mathcal{S}em_{M\bullet^+}(\mathcal{AL})$ based on the goal-independent structure sharing semantics $\mathcal{S}em_{M\star}(\wp(\overline{\mathcal{SD}}_{VI}))$. The former uses the specific liveness related auxiliary operations, *i.e.*, $\mathsf{init}^{\mathcal{AL}}$, $\mathsf{comb}^{\mathcal{AL}}$, $\mathsf{add}^{\mathcal{AL}}$ and $\mathsf{combupdate}^{\mathcal{AL}}$ (See Definition 8.21), while the latter uses the structure sharing related auxiliary operations, *i.e.*, init_a, comb_a and add_a (See Definition 6.32).

The complete definition of $\mathcal{S}em_{M\bullet^+}$, immediately instantiated with the do-

main $\mathcal{AL}$, is depicted in Figure 10.1. We discuss some of the particularities of this definition.

- We already instantiated these semantics with the auxiliary operations for abstract liveness descriptions.
- We assume that $e^\star$, the goal-independent rule base meaning w.r.t. structure sharing, also contains the local structure sharing components of the procedures that are defined in other modules than the module that is actually analysed. Formally, this information is implicitly included in the definition of the goal-independent part of the semantics ($\mathcal{S}em_{M\star}(\wp(\overline{\mathcal{SD}}_{VI}))$). In practice, for each module this information will normally be recorded in so called interface files. These files are then consulted at the start of the compilation or analysis of modules depending on them.
- We assume that for these semantics procedures imported from other modules are available as procedures with an empty body. This view enables us to keep the same semantic rule for procedure calls, independent of whether these are calls to procedures defined in the same module or procedures defined elsewhere;
- In the rule defining the semantics of programs, we explicitly expand the goal-independent rule base meaning from pure structure sharing to liveness descriptions. This avoids the need for redefining the auxiliary operations using information from that rule base meaning, and allows us to use the usual auxiliary functions defined for the abstract liveness description domain $\mathcal{AL}$. For this purpose we overload the definition of $\mathsf{augment}_a$ defined by Equation (8.4) on page 180 and write $\mathsf{augment}_a * (\mathbf{R}_{M\star}[\![r]\!])$ where we use $\mathsf{augment}_a$ as a shorthand notation for augmenting each of the entries in that rulebase meaning to elements of $\mathcal{AL}$, hence obtaining a rulebase meaning in terms of abstract liveness descriptions, instead of structure sharing alone. This allows us to write the semantics of a Mercury program as depicted in Figure 10.1: based on the augmented goal-independent rule-base meaning of the program, the goal-dependent semantics of that program is given by the semantics of the query in that program, using $\mathsf{init}^{\mathcal{AL}}$ as the initial call description of that query.

It can easily be shown that $\mathcal{S}em_{M\delta^+}(\mathcal{AL})$ is equivalent to $\mathcal{S}em_{M\bullet^+}(\mathcal{AL})$. Indeed, $\mathcal{S}em_{M\bullet}$ has been augmented in exactly the same was as $\mathcal{S}em_{M\delta}$ was augmented to obtain $\mathcal{S}em_{M\delta^+}$. Moreover, the goal-independent based part could have been instantiated with $\mathcal{AL}$ instead of with structure sharing alone, therefore definitely guaranteeing equivalence, yet we argued that threading liveness information in the goal-independent context is simply unnecessary, and therefore deriving structure sharing information alone in that setting suffices.

$$
\begin{array}{lcl}
\mathbf{P}_{M^{\bullet+}}[\![r;q]\!] & = & \text{let } \iota = \mathsf{init}^{\mathcal{AC}} \text{ in} \\
 & & \text{let } e^\star = \mathsf{augment}_a(\mathbf{R}_{M\star}[\![r]\!]) \text{ in} \\
 & & \quad \mathbf{G}_{M^{\bullet+}}[\![q]\!](\mathbf{R}_{M^{\bullet+}}[\![r]\!]e^\star)\iota\,\iota \\
\mathbf{R}_{M^{\bullet+}}[\![r]\!]e^\star & = & \mathrm{fix}(\mathbf{F}_{M^{\bullet+}}[\![r]\!]e^\star) \\
\mathbf{F}_{M^{\bullet+}}[\![p_1 \ldots p_i \ldots p_{n_p}]\!]e^\star(e,A)p_i(\overline{Y})\mathcal{S} & & \\
 & = & \mathbf{Pr}_{M^{\bullet+}}[\![p_i]\!]e^\star(e,A)p_i(\overline{Y})\mathcal{S} \\
\mathbf{Pr}_{M^{\bullet+}}[\![h \leftarrow g]\!]e^\star(e,A)a\mathcal{S} & = & \text{let } \mathcal{S}_g = \rho_{a\to h}\,((\mathcal{S})|_a) \text{ in} \\
 & & \text{let } (\mathcal{S}_1, A_1) = \mathbf{G}_{M^{\bullet+}}[\![g]\!](e,A)\mathcal{S}_g\mathsf{init}_{h\leftarrow g} \text{ in} \\
 & & \quad (e^\star a, A_1) \\
\mathbf{G}_{M^{\bullet+}}[\![g_1,g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M^{\bullet+}}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad \mathbf{G}_{M^{\bullet+}}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1} \\
\mathbf{G}_{M^{\bullet+}}[\![g_1;g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M^{\bullet+}}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M^{\bullet+}}[\![g_2]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_{l_1} \sqcup \mathcal{S}_{l_2}, \mathsf{merge}(A_1,A_2)) \\
\mathbf{G}_{M^{\bullet+}}[\![\text{if } g_1 \text{ then } g_2 \text{ else } g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l & & \\
 & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M^{\bullet+}}[\![g_1]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_2}, A_2) = \mathbf{G}_{M^{\bullet+}}[\![g_2]\!](e,A_1)\mathcal{S}_g\mathcal{S}_{l_1} \text{ in} \\
 & & \text{let } (\mathcal{S}_{l_3}, A_3) = \mathbf{G}_{M^{\bullet+}}[\![g_3]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_{l_2} \sqcup \mathcal{S}_{l_3}, \mathsf{merge}(A_2,A_3)) \\
\mathbf{G}_{M^{\bullet+}}[\![\text{not } g]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } (\mathcal{S}_{l_1}, A_1) = \mathbf{G}_{M^{\bullet+}}[\![g]\!](e,A)\mathcal{S}_g\mathcal{S}_l \text{ in} \\
 & & \quad (\mathcal{S}_l, A_1) \\
\mathbf{G}_{M^{\bullet+}}[\![l]\!](e,A)\mathcal{S}_g\mathcal{S}_l & = & \text{let } \mathcal{S}'_g = \mathsf{combupdate}^{\mathcal{AC}}(\mathsf{pp}(l), \mathcal{S}_g, \mathcal{S}_l) \text{ in} \\
 & & \quad (\mathbf{L}_{M^{\bullet+}}[\![l]\!]\, e\, \mathcal{S}'_g\mathcal{S}_l, \\
 & & \qquad A[(\mathsf{pp}(l), \mathcal{S}_g), \mathcal{S}'_g]) \\
\mathbf{L}_{M^{\bullet+}}[\![\text{unif}]\!]\, e\, \mathcal{S}'_g\mathcal{S}_l & = & \mathsf{add}^{\mathcal{AC}}(\text{unif}, \mathcal{S}_l) \\
\mathbf{L}_{M^{\bullet+}}[\![p(\overline{X})]\!]\, e\, \mathcal{S}'_g\mathcal{S}_l & = & \mathsf{comb}^{\mathcal{AC}}(\mathcal{S}_l, e(p(\overline{X}), \mathcal{S}'_g))
\end{array}
$$

Figure 10.1: Augmented goal-independent based semantics $\mathcal{S}em_{M^{\bullet+}}$. Here $e^\star$ represents the goal-independent rule-base meaning in terms of the structure sharing information, yet each single structure sharing element is augmented to a tuple fitting the definition of a liveness description.

In Section 10.5.3 we detail some of the implementation issues.

Note that in this setting we tacitly assume that backward use is not a problem in the modularisation process. Indeed, whether in the simplified definition as well as in the analysis-based one, backward use remains a goal-independent property, and therefore easily adaptable to a module-based setting. In the case of the analysis-based definition, we can make use of the optimisation interface files to record the backward use information for each (exported) procedure.

10.3 Modular Reuse Analysis

In the Melbourne Mercury compiler, each module of a program is compiled separately. There are several problems for reuse analysis involved with this scheme:

1. When compiling a given module, how do we detect opportunities for reuse without knowing the exact liveness call descriptions?
2. Suppose we do know how to detect such potential opportunities, then we need to know which reuse versions are worthwhile to be generated without risking code explosion.
3. And finally, suppose we have generated different versions for the different combinations of reuse opportunities of the procedures defined in a module, how do we check when it is safe to call such an optimised version of that procedure?

10.3.1 Detection of Reuse Opportunities

A useful property of reuse is that the number of reuse opportunities in a procedure decreases monotonically with increasing liveness call descriptions, *i.e.*, with increasing initial liveness and increasing structure sharing. Hence, performing a call dependent analysis of a procedure p defined in the interface of a module m starting with a call description with an empty initial liveness set and an empty structure sharing set guarantees that the analysis results in a description of *all* the direct reuses possible within that procedure[1]. If the call graph of p contains only calls to procedures defined in the same module m, then all indirect reuses can also be spotted. If p calls a number of procedures defined elsewhere, then we first need a tactic to verify the reuses in those procedures. This aspect is handled in Section 10.3.3.

This view calls for a goal-independent liveness analysis where indeed initial calls are set to be calls with call description $\langle \{\}, \{\} \rangle$ (this corresponds to the definition of $\mathsf{init}^{\mathcal{AL}}$ in Definition 8.19). In the following sections, we call this description the *empty call description* of a procedure.

[1] Of course, within the limits of the precision of the analysis.

10.3.2 Generating Reuse Versions

If we want to strictly follow the compilation rules of the Melbourne Mercury compiler, meaning that all modules are compiled separately and are compiled only once, then we have no clue of how procedures will be called w.r.t. liveness information. In such circumstances we can only use heuristics to decide what versions are worthwhile to generate. These heuristics can be:

- *do not generate any reuse version*. In such case there will be no reuse of course, which is not the purpose of our analysis;
- *generate all possible combinations of direct reuses and indirect reuses*. If a procedure has n deconstructions making direct reuse possible, and m calls to procedures with some reuse, then in theory, we can produce at least 2^{n+m} different reuse versions for that procedure, even more if we produce different versions for each of the versions of the called procedures. For n and m big, such strategy yields to code explosion, an unwanted effect.
- *generate a limited number of versions*. And here, the simplest approach is to generate at most two versions per procedure: one version that contains no reuse at all (or as we will later see, only "*unconditional*" reuses), and one version that allows the most reuses of all. In a later section we come back to this approach, and discuss some variants of it.

In our first prototype and first implementation into the Melbourne Mercury compiler we opted for generating at most two versions.

10.3.3 Safe Calls to Reuse Versions

Consider again the deterministic procedure of `append` from Example 4.4, of which the definition is repeated below:

```
%:- pred append(list(T),list(T),list(T)).
%:- mode append(in,in,out) is det.
append(X,Y,Z):-
    (
        (1) X == [],
        (2) Z := Y
    ;
        (3) X => [Xe|Xs],
        (4) append(Xs,Y,Zs),
        (5) Z <= [Xe|Zs]
    ).
```

Suppose we analyse `append` w.r.t. the empty call description. At the deconstruction `X => [Xe|Xs]`, the current liveness component is the set $\{Y^{\overline{e}}, Xe^{\overline{e}}, Xs^{\overline{e}}\}$,

hence $X^{\overline{\epsilon}}$ is available for reuse. Together with the construction unification at program point (5), the deconstruction forms a perfect matching deconstruction/construction pair, thus we have possible direct reuse. The recursive call is called with the (projected and renamed) call description $\langle\{\,\}, \{Z^{\overline{\epsilon}}\}\rangle$ (there is indeed no sharing for the input variables Xs and Y, and only Zs —thus Z after renaming—is live outside the call). And also for this call we can verify that the deconstructed list-cells of the first list are available for reuse.

Thus, we discover that append has direct reuse and indirect reuse, and we generate two versions for that procedure: a version without reuse, and a version that completely reuses the top level list cells of the first list for creating the new list Z.

Now we can argue: given a procedure p with a procedure call to append in it, with say a call description AL, can we generate a version for p with a call to the reuse version of append? Is it a safe reuse? For all calls to p? How do we verify this?

In a first approach, one might use the empty call description as a measure: for every call description AL that is subsumed by the call description with which the procedure was analysed, in this case the empty call description, the reuse results can be trusted. Although this is a very easy and certainly safe test, it is too strict and only allows a limited number of calls to reuse versions of the analysed procedures. Indeed, in the case of append, we can verify that even if the elements of the first list are live in the calling environment (hence the call description is not subsumed by the empty call description), then the reuse version is still safe to call. Similarly if the second list is live, then reuse remains safe. Yet again the call description is not subsumed by the empty call description with which the reuses in append can be discovered. More generally, in the case of append we can intuitively see that reuse in append is always safe as long as the backbone of the first list is not live in the calling environment. It is this kind of results that we would like to obtain in our analysis.

The general problem to be addressed is: which results of the goal-independent liveness analysis need to be stored and how do we use these results to check that a particular call meets the requirements to call the reuse version resulting from these call dependent analyses? For this purpose we introduce the notions of *reuse information* and *conditions for reuse*. We introduce these notions in the context of direct reuses (Section 10.3.4), and then demonstrate how they *propagate* through indirect reuses (Section 10.3.5).

As the following sections only deal with abstract values, we drop the convention of over-lining the involved components. Over-lining will be used for other purposes.

10.3.4 Reuse Information: Direct Reuse

We investigate the situation of a call to a procedure with possible direct reuse.

Let q be a procedure defined in a module m_1:

```
%:- pred q(...).
%:- mode q(...).
q(Q1, ... ,Qn) :- ..., (i) X => f(...), ... .
```

We assume that a goal-independent structure sharing analysis was performed. Let $A_{l,i}$ be the local structure sharing at program point (i) in procedure q/n. Let U_i be the union of the data structures in forward use at program point (i) and the data structures in backward use at (i), *i.e.*, $U_i = \mathsf{data}_a(\mathsf{forward}(i)) \sqcup_{ad} \mathsf{data}_a(\mathsf{backward}(i))$. Let $\mathcal{H} = \{Q_1, \ldots, Q_n\}$ be the set of head variables of q/n.

We assume that q/n has an opportunity for direct reuse. This means that during the goal-dependent analysis of q/n w.r.t. the empty call description $\langle\{\,\},\{\,\}\rangle$, the top level data structure of X was discovered to become available for reuse. Given the choice of our version-generation strategy, we compile q/n into a version without reuse, and a version realising the detected reuse possibility.

Now assume a call to q/n from another procedure, say p/m, defined in a module m_2. During the reuse analysis of p/m in m_2 we want to know whether replacing the call to q/n by a call to the reuse version of q/n is safe, yet without redoing the full analysis of q/n. Suppose that q/n is called with call description $AL = \langle A_0, L_0\rangle$, then the question is: can $X^{\overline{e}}$ still be reused?

Let us do the analysis of q/n with call description AL, then the current liveness component of the liveness description at program point (i) is given by Equation (8.6), *i.e.*:

$$\begin{array}{rcl} L_i & = & \mathsf{live}_a(i, A_0, A_{l,i}, L_0) \\ & = & \mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})) \end{array} \tag{10.1}$$

If we want to check whether $X^{\overline{e}}$ becomes available for reuse at (i) for that particular call, then we need to verify that $X^{\overline{e}} \not\leq_{ad} L_i$.

Hence, to verify that reuse is still safe, we need to compute L_i. To do that computation, we only need the following information at program point (i):

1. the data structures in forward and backward use at program point (i), U_i;
2. the local component for structure sharing, $A_{l,i}$,
3. and of course the data structure that should become available for reuse, $X^{\overline{e}}$.

We call this trio the *reuse information* for verifying a case of direct reuse, and denote it by the tuple $\langle\!\langle \{X^{\overline{e}}\}, U_i, A_{l,i}\rangle\!\rangle$. Verifying reuse for a specific call description $AL = \langle A_0, L_0\rangle$ comes down to verifying the expression:

$$X^{\overline{e}} \not\leq_{ad} \mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})) \tag{10.2}$$

We call Equation (10.2) the *reuse condition* for reusing $X^{\overline{\epsilon}}$.

If D represents a set of data structures, all pointing to the same heap cells to be reused[2], then we generalise these notions as follows:

Definition 10.1 (Reuse Information, Reuse Condition) *Let q/n be a procedure with possible reuse at program point (i), then $R_i = \langle\!\langle D_i, U_i, A_{l,i} \rangle\!\rangle$ with D_i the set of abstract data structures referring to the reusable heap cells[3], U_i the data structures in forward and backward use at (i), and $A_{l,i}$ the local component of structure sharing, is called the* reuse information *concerning the reuse of the heap cells referred to by D_i.*

Let $AL = \langle A_0, L_0 \rangle$ be a call description for q/n, then verifying reuse for AL is described by:

$$\forall X^{\overline{s}} <_{ad} D : X^{\overline{s}} \not<_{ad} \mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})) \qquad (10.3)$$

called the reuse condition *for reusing the structure pointed at by the set of data structures D. If the data structure set in AL meets the reuse condition expressed by Equation (10.3) w.r.t. the reuse information R_i, then this is denoted by*

$$AL \sharp R_i$$

If a procedure contains a number of matching deconstruction/construction pairs, and each of these pairs allows direct reuse for the default call description, then each of the involved deconstruction unifications produces its own reuse information tuple. According to our version generation strategy we produce a version realising all the detected reuses. To verify the safeness of using such a version, all the reuse information tuples need to be verified. Thus, suppose a procedure has n opportunities for reuse, with reuse information tuples $R_1, \ldots R_n$ (note that here the subscripts refer to the number of the tuple in the sequence and not the program point to which it belongs), then calling the reuse version for a call with call description AL is safe iff $AL \sharp R_i$ for all $1 \leq i \leq n$.

In a first approach, it therefore suffices to store the reuse information tuples for each procedure in an appropriate interface file for the module in which the procedures are defined, and each time reuse needs to be verified, we recompute Equation (10.2) for each specific liveness call description AL and each reuse information. This approach is feasible if the sets of the variables in forward use, backward use, but mainly the set of structure sharing would be limited in size. In a realistic setting, the latter set can count up to 500 or even more pairs of sharing data structures due to a large number of variables but also to large deep type graphs (See Section 11.3). The extend_a operation is also a complex operation, so not only do these large sets mean large interface files, and larger memory requirements for the compiler, but also larger computation times have to be faced. Given

[2]Such a set of data structures is obtained due to structure sharing, and always represents one and the same term as kept on the heap.

[3]At this moment D is always a singleton set, yet this will change in later sections.

the overall complexity of the analyses, especially the computation times need to be acceptable.

We therefore need to compact the reuse information. The intuition is as follows. Each of the components in a call description $\langle A_0, L_0 \rangle$ only relates to head variables of the concerned procedure. Therefore, they can only affect the liveness of other head variables. Hence, if we manage to *translate* the reuse information in such a way that also only head variables are concerned, then we have a pure head variable related property. As the set of variables is limited to the head variables, one can expect that the reuse information will be more compact. And this is the property we want to achieve. On the following pages we will show that translating the reuse information by extending it w.r.t. the local structure sharing and then projecting the result on the head variables of the concerned procedure guarantees correct results w.r.t. the reuses that are safely allowed. The intended translation of a reuse information tuple of the direct reuse of a data structure $X^{\overline{\epsilon}}$, $\langle\!\langle \{X^{\overline{\epsilon}}\}, U_i, A_{l,i} \rangle\!\rangle$, looks as follows:

$$\begin{array}{ccc} \{X^{\overline{\epsilon}}\} & \Rightarrow & \left(\mathsf{extend}_a(\{X^{\overline{\epsilon}}\}, A_{l,i})\right)|_{\mathcal{H}} \\ U_i & \Rightarrow & (\mathsf{extend}_a(U_i, A_{l,i}))|_{\mathcal{H}} \\ A_{l,i} & \Rightarrow & (A_{l,i})|_{\mathcal{H}} \end{array}$$

Before demonstrating that we can safely translate the reuse information in this way to the head variables, we show a number of properties of the local structure sharing information $A_{l,i}$ w.r.t. a structure sharing call description A_0, and their combination.

Lemma 10.1 *Let $A_{l,i}$ be the local component for structure sharing at a program point (i) of a procedure q and A_0 the structure sharing call description of that procedure*[4]*. Let $\mathcal{H}$ denote the head variables of the procedure to which the structure sharing information relates, and let $\mathcal{H}_{\mathtt{in}}$ be the subset of these head variables containing only the input variables of the procedure,*i.e., *$\mathcal{H}_{\mathtt{in}} = \mathtt{in}(q)$. We have the following properties:*

1. *$Vars(A_0) \subseteq \mathcal{H}_{\mathtt{in}}$*
2. *For all abstract structure sharing pairs $(\alpha - \beta)$:*

$$\begin{array}{c} Vars((\alpha - \beta)) \subseteq \mathcal{H}_{\mathtt{in}} \text{ and } (\alpha - \beta) \leq_a \mathsf{comb}_a(A_0, A_{l,i}) \\ \Updownarrow \\ (\alpha - \beta) \leq_a A_0 \end{array}$$

Indeed, when a procedure is called, structure sharing can only exist between input arguments of the procedure. The second property states that a procedure can

[4]We assume that both values are seen in the context of the differential semantics $\mathcal{S}em_{M\delta\wp}(\overline{\mathcal{SD}}_{VI})$ or related ones.

not add structure sharing relations between variables that are already instantiated when the procedure is called.

We also need to go into the details of the alternating closure operation. We introduce a new operation $\mathsf{altclos}_i$ that returns the edges of paths of length i, and $\mathsf{altclos}_{>i}$ for paths of length $> i$.

Definition 10.2 ($\mathsf{altclos}_i$, $\mathsf{altclos}_{>i}$) *Given the sets of unordered pairs A and B, then $\mathsf{altclos}_i(A,B)$, where $i > 0$ returns the set of unordered pairs for which there exists a path of length i alternating between pairs of A and B. $\mathsf{altclos}_{>i}(A,B)$ returns the set of pairs for which there exists a path of length greater than i alternating between elements from A and B. Formally:*

$$\begin{array}{lcl} \mathsf{altclos}_i(A,B) & = & \left\{ (a_0,a_i) \;\middle|\; \begin{array}{l} (a_0,a_1)\cdot(a_1,a_2)\cdot\ldots\cdot(a_{i-1},a_i) \\ \quad \textit{over } A \textit{ and } B, \\ \quad \textit{such that } \forall (a_{j-1},a_j), 1 \leq j < i : \\ \qquad \left\{ \begin{array}{l} (a_{j-1},a_j) \in X \Rightarrow (a_j,a_{j+1}) \in Y \\ (a_{j-1},a_j) \in Y \Rightarrow (a_j,a_{j+1}) \in X \end{array} \right. \end{array} \right\} \\ \mathsf{altclos}_{>i}(A,B) & = & \bigcup_{j>i} \mathsf{altclos}_j(A,B) \end{array}$$

Note that $\mathsf{altclos}_1(A,B) = A \cup B$.

We add a *directional* definition of the paths formed in an alternating closure operation.

Definition 10.3 (Directional $\mathsf{altclos}_{i\rightarrow}$, and $\mathsf{altclos}_{i\leftarrow}$) *The* right-directional alternating closure *of paths of length i over the sets of pairs A and B is defined as the tuples that are obtained by forming paths of length i of elements alternating between elements from A and B, such that each path* starts *by a vertex in A. This set is denoted by $\mathsf{altclos}_{i\rightarrow}(A,B)$. We define the* left-directional alternating closure *in a similar way, yet here all paths start in the second set. We denote it by $\mathsf{altclos}_{i\leftarrow}(A,B)$.*

We can show that $\mathsf{altclos}_i(A,B) = \mathsf{altclos}_{i\rightarrow}(A,B) \cup \mathsf{altclos}_{i\leftarrow}(A,B)$, and that if i is an even natural number, then $\mathsf{altclos}_{i\rightarrow}(A,B) = \mathsf{altclos}_{i\leftarrow}(A,B)$ for all sets of pairs A and B.

We redefine $\mathsf{altclos}(A,B)$ as follows:

$$\mathsf{altclos}(A,B) = \mathsf{altclos}_1(A,B) \cup \mathsf{altclos}_2(A,B) \cup \ldots \tag{10.4}$$

We now show an important property of the combination operation for abstract structure sharing sets.

Lemma 10.2 *Consider a procedure q (with head variables $\mathcal{H}$, and input $\mathcal{H}_{\mathtt{in}}$) with structure sharing call description A_0 and local structure sharing description at program point i given by $A_{l,i}$, again assuming the differential semantics as our context. Let $TA_0 = \mathsf{termshift}(A_0)$ and $TA_{l.i} = \mathsf{termshift}(A_{l,i})$, then*

$$(\mathsf{altclos}(TA_0, TA_{l,i}))|_{\mathcal{H}} = \mathsf{altclos}(TA_0,\ (TA_{l,i})|_{\mathcal{H}}) \tag{10.5}$$

Proof Using Equation (10.4), we have

$$\mathsf{altclos}(TA_0, TA_{l,i}) = \bigcup_{j \geq 1} \mathsf{altclos}_j(TA_0, TA_{l,i})$$

We show that Equation (10.5) holds for each $\mathsf{altclos}_j(TA_0, TA_{l,i})$, $j \geq 1$:

- $j = 1$: $(\mathsf{altclos}_1(TA_0, TA_{l,i}))|_{\mathcal{H}} = (TA_0 \cup TA_{l,i})|_{\mathcal{H}} = (TA_0)|_{\mathcal{H}} \cup (TA_{l,i})|_{\mathcal{H}} = TA_0 \cup (TA_{l,i})|_{\mathcal{H}} = \mathsf{altclos}_1(TA_0, (TA_{l,i})|_{\mathcal{H}})$.
- $j = 2$:

$$\begin{array}{rl} (\mathsf{altclos}_2(TA_0, TA_{l,i}))|_{\mathcal{H}} = & (\mathsf{altclos}_{2\rightarrow}(TA_0, TA_{l,i}))|_{\mathcal{H}} \\ = & (\{(\alpha - \gamma) \mid (\alpha - \beta) \in TA_0 \wedge (\beta - \gamma) \in TA_{l,i}\})|_{\mathcal{H}} \\ = & \{(\alpha - \gamma) \mid (\alpha - \beta) \in (TA_0)|_{\mathcal{H}} \wedge (\beta - \gamma) \in (TA_{l,i})|_{\mathcal{H}}\} \\ = & \{(\alpha - \gamma) \mid (\alpha - \beta) \in TA_0 \wedge (\beta - \gamma) \in (TA_{l,i})|_{\mathcal{H}}\} \\ = & \mathsf{altclos}_2(TA_0, (TA_{l,i})|_{\mathcal{H}}) \end{array}$$

- $j = 3$: $\mathsf{altclos}_3(TA_0, TA_{l,i}) = \mathsf{altclos}_{3\rightarrow}(TA_0, TA_{l,i}) \cup \mathsf{altclos}_{3\leftarrow}(TA_0, TA_{l,i})$. Elements in $\mathsf{altclos}_{3\rightarrow}(TA_0, TA_{l,i})$ are formed by constructing paths $(\alpha - \beta) \cdot (\beta - \gamma) \cdot (\gamma - \delta)$ where $(\alpha - \beta), (\gamma - \delta) \in TA_0$ and $(\beta - \gamma) \in TA_{l,i}$. Knowing that $Vars(TA_0) \subseteq \mathcal{H}_{\mathtt{in}}$, implies that $Vars((\beta - \gamma)) \subseteq \mathcal{H}_{\mathtt{in}}$, yet we know that this is not possible given the fact that a procedure can not add structure sharing between input variables (Lemma 10.1). Hence $\mathsf{altclos}_{3\rightarrow}(TA_0, TA_{l,i}) = \{\}$. Therefore:

$$\begin{array}{rl} (\mathsf{altclos}_3(TA_0, TA_{l,i}))|_{\mathcal{H}} = & (\mathsf{altclos}_{3\leftarrow}(TA_0, TA_{l,i}))|_{\mathcal{H}} \\ = & \{(\alpha - \delta) \mid (\alpha - \beta), (\gamma - \delta) \in (TA_{l,i})|_{\mathcal{H}} \wedge (\beta - \gamma) \in (TA_0)|_{\mathcal{H}}\} \\ = & \{(\alpha - \delta) \mid (\alpha - \beta), (\gamma - \delta) \in (TA_{l,i})|_{\mathcal{H}} \wedge (\beta - \gamma) \in TA_0\} \\ = & \mathsf{altclos}_3(TA_0, (TA_{l,i})|_{\mathcal{H}}) \end{array}$$

- $j > 3$: In each path of length > 3 alternating between elements from TA_0 and $TA_{l,i}$ we always encounter the sub path of length three $(\alpha - \beta) \cdot (\beta - \gamma) \cdot (\gamma - \delta)$ where $(\alpha - \beta), (\gamma - \delta) \in TA_0$ and $(\beta - \gamma) \in TA_{l,i}$. Such paths would imply that $Vars((\beta - \gamma)) \subseteq \mathcal{H}_{\mathtt{in}}$ which is impossible. Hence $\mathsf{altclos}_{>3}(TA_0, TA_{l,i}) = \{\}$.

This case study proves that projecting the local structure sharing set onto the head variables before combining it with the call description yields the same result as performing the combination first, and only then do the projection. Thus: $(\mathsf{comb}_a(A_0, A_{l,i}))|_{\mathcal{H}} = \mathsf{comb}_a(A_0, (A_{l,i})|_{\mathcal{H}})$.

□

As a side-effect of proving the previous lemma we have the following corollary:

Corollary 10.1 $\mathsf{altclos}_{3\rightarrow}(TA_0, TA_{l,i}) = \{\,\}$, *and* $\mathsf{altclos}_{>3}(TA_0, TA_{l,i}) = \{\,\}$, *thus*

$$\begin{array}{rcl} \mathsf{altclos}(TA_0, TA_{l,i}) & = & \mathsf{altclos}_1(TA_0, TA_{l,i}) \\ & & \cup\mathsf{altclos}_2(TA_0, TA_{l,i}) \\ & & \cup\mathsf{altclos}_{3\leftarrow}(TA_0, TA_{l,i}) \end{array}$$

Using the previous lemma's we move on to our main theorem that allows us to compact the information needed for verifying reuse information that is only related to the head variables of the procedures. This means a significant reduction in information to be stored and computation time to be spent.

Definition 10.4 (Compacted Reuse Information) *Let $R_i = \langle\!\langle D_i, U_i, A_{l,i} \rangle\!\rangle$ be the reuse information at a program point i in a procedure q/n with head variables $\mathcal{H}$, then*

$$\begin{array}{rcl} \mathsf{compact}(R_i, \mathcal{H}) & = & \langle\!\langle \mathsf{compact}(D_i, \mathcal{H}), \mathsf{compact}(U_i, \mathcal{H}), \mathsf{compact}(A_{l,i}, \mathcal{H}) \rangle\!\rangle \\ & = & \langle\!\langle \overline{D_i}, \overline{U_i}, \overline{A_{l,i}} \rangle\!\rangle \end{array}$$

where

$$\begin{array}{rcl} \overline{D_i} & = & (\mathsf{extend}_a(D_i, A_{l,i}))|_{\mathcal{H}} \\ \overline{U_i} & = & (\mathsf{extend}_a(U_i, A_{l,i}))|_{\mathcal{H}} \\ \overline{A_{l,i}} & = & (A_{l,i})|_{\mathcal{H}} \end{array}$$

The result of $\mathsf{compact}(R_i, \mathcal{H})$ *is called the* compacted reuse information *of R_i w.r.t. the head variables $\mathcal{H}$.*

The compacted reuse information of a tuple R is usually denoted as $\overline{R}$, similarly for each of its components. We sometimes apply the compaction operation on individual components of a reuse information tuple, assuming that the local structure sharing component with which the extension operation is performed is clear from the context. Therefore, if $R = \langle\!\langle D, U, A \rangle\!\rangle$, we sometimes simply write $\mathsf{compact}(D, \mathcal{H})$, $\mathsf{compact}(U, \mathcal{H})$, and $\mathsf{compact}(A, \mathcal{H})$.

Theorem 10.1 *Consider a procedure q/n with $\mathcal{H}$ the set of head variables of q/n, and R_i the reuse information at a program point (i). Let $AL_{0,\delta}$ be the empty call description for q, and let AL be some other valid call description for q. If $AL_{0,\delta}$ meets the reuse condition w.r.t. R_i, and AL meets the reuse condition w.r.t. $\overline{R_i} = \mathsf{compact}(R_i, \mathcal{H})$, the compacted reuse information, then AL also meets the reuse condition w.r.t. R_i, hence reuse for AL is safe. Formally:*

$$\left(AL_{0,\delta} \sharp R_i \wedge AL \sharp \overline{R_i}\right) \Rightarrow AL \sharp R_i \tag{10.6}$$

Proof Let $R_i = \langle\!\langle D_i, U_i, A_{l,i} \rangle\!\rangle$, and $\overline{R_i} = \langle\!\langle \overline{D_i}, \overline{U_i}, \overline{A_{l,i}} \rangle\!\rangle$. Let $AL_{0,\delta} = \langle \{\,\}, \{\,\} \rangle$ — the empty call description, and $AL = \langle A_0, L_0 \rangle$. Then $AL_{0,\delta} \sharp R_i$ is equivalent[5] to

$$\forall \alpha <_{ad} D_i : \alpha \not<_{ad} \mathsf{extend}_a(U_i, A_{l,i}) \tag{10.7}$$

[5]Indeed, $\mathsf{extend}_a(\{\,\}, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(\{\,\}, A_{l,i})) = \mathsf{extend}_a(U_i, A_{l,i})$.

and $AL\sharp\overline{R_i}$ is the same as

$$\forall\beta <_{ad} \overline{D_i} : \beta \not<_{ad} \mathsf{extend}_a(L_0, \overline{A_{l,i}}) \sqcup_{ad} \mathsf{extend}_a(\overline{U_i}, \mathsf{comb}_a(A_0, \overline{A_{l,i}})) \tag{10.8}$$

Finally, verifying the reuse condition $AL\sharp R_i$ means verifying the equation

$$\forall\gamma <_{ad} D_i : \gamma \not<_{ad} \mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})) \tag{10.9}$$

Let $\alpha <_{ad} D_i$, we prove the theorem by showing that the assumption

$$\alpha <_{ad} \mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})) \tag{10.10}$$

violates one of the conditions $AL_{0,\delta}\sharp R_i$, and $AL\sharp\overline{R_i}$, hence Equations (10.7) and (10.8). We perform a case study.

Let $TA_0 = \mathsf{termshift}(A_0)$ and $TA_{l,i} = \mathsf{termshift}(A_{l,i})$.

The assumption expressed by Equation 10.10 is true if either α is subsumed by $\mathsf{extend}_a(L_0, A_{l,i})$ or by $\mathsf{extend}_a(U_i, \mathsf{comb}(A_0, A_{l,i}))$.

1. $\alpha <_{ad} \mathsf{extend}_a(L_0, A_{l,i})$:
   - 1.1. $\alpha <_{ad} L_0$. As $Var(\alpha) \in \mathcal{H}$, $\alpha <_{ad} \overline{D_i}$, hence α is subsumed by $\mathsf{extend}_a(L_0, \overline{A_{l,i}})$, which contradicts Equation (10.8).
   - 1.2. $\exists\beta <_{ad} L_0 : (\alpha - \beta) \leq_a A_{l.i}$. As $Vars(L_0) \subseteq \mathcal{H}$, then $Var(\beta) \in \mathcal{H}$, thus $\beta <_{ad} \overline{D_i}$. Given $\beta <_{ad} L_0$, then also $\beta <_{ad} \mathsf{extend}_a(L_0, A), \forall A$, thus also $\beta <_{ad} \mathsf{extend}_a(L_0, \overline{A_{l,i}})$. This means that β is subsumed by the expression on the right-hand side of Equation (10.8), hence contradicts that equation.
2. $\alpha <_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}(A_0, A_{l,i}))$.
   - 2.1. $\alpha <_{ad} U_i$. This of course contradicts Equation (10.7).
   - 2.2. $\exists\beta <_{ad} U_i : (\alpha - \beta) \leq_a \mathsf{comb}(A_0, A_{l,i})$. For this situation we perform a case study on whether β or α are data structures of a head variable or not.
     - 2.2.1. $Var(\beta) \in \mathcal{H}, Var(\alpha) \in \mathcal{H}$.
       $Var(\alpha) \in \mathcal{H}$ implies that $\alpha <_{ad} \overline{D_i}$. $Var(\beta) \in \mathcal{H}$ means that $\beta <_{ad} \overline{U_i}$. Knowing that both α and β relate to head variables also means that $(\alpha - \beta) \leq_a (\mathsf{comb}(A_0, A_{l,i}))|_{\mathcal{H}}$ hence $(\alpha - \beta) \leq_a \mathsf{comb}(A_0, (A_{l,i})|_{\mathcal{H}})$ (Lemma 10.2) where $(A_{l,i})|_{\mathcal{H}}$ is simply the compacted local structure sharing, *i.e.*, $\overline{A_{l,i}}$. Combining all these properties we obtain that $\alpha <_{ad} \overline{D_i}$ and $\alpha <_{ad} \mathsf{extend}_a(\overline{U_i}, \mathsf{comb}_a(A_0, \overline{A_{l,i}}))$, which contradicts Equation (10.8).

2.2.2. $Var(\beta) \in \mathcal{H}, Var(\alpha) \notin \mathcal{H}$.
Either $(\alpha - \beta)$ is formed by paths of length 1, or by paths of length $j > 1$.

- $(\alpha - \beta) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. As $Vars(TA_0) \subseteq \mathcal{H}$ while $Var(\alpha) \notin \mathcal{H}$, then $(\alpha - \beta) \in TA_{l,i}$, *i.e.*, $(\alpha - \beta) \leq_a A_{l,i}$. With $\beta \in U_i$ this contradicts Equation (10.7).
- $(\alpha - \beta) \in \mathsf{altclos}_j(TA_0, TA_{l,i})$, $j > 1$. Then $\exists \gamma : (\alpha - \gamma) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. As $Var(\alpha) \notin \mathcal{H}$, $(\alpha - \gamma) \in TA_{l,i}$, and thus $(\gamma - \beta) \in \mathsf{altclos}_{(j-1)\rightarrow}(TA_0, TA_{l,i})$ which means that $Var(\gamma) \in \mathcal{H}$. As $(\alpha - \gamma) \in TA_{l,i}$, then $\gamma \in \overline{D_i}$. With both $Var(\gamma) \in \mathcal{H}$ and $Var(\beta) \in \mathcal{H}$ we can repeat the reasoning of Item 2.2.1., hence obtaining a contradiction with Equation (10.8).

2.2.3. $Var(\beta) \notin \mathcal{H}, Vars(\alpha) \in \mathcal{H}$.
We repeat almost the same reasoning as for Item 2.2.2.: $(\alpha - \beta)$ is either in $\mathsf{altclos}_1(TA_0, TA_{l,i})$, hence is formed by paths of length 1, or in $(\alpha - \beta) \in \mathsf{altclos}_j(TA_0, TA_{l,i})$, $j > 1$, which means that the paths are of length $j > 1$.

- $(\alpha - \beta) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. As $Var(\beta) \notin \mathcal{H}$: $(\alpha - \beta) \in TA_{l,i}$. With $\beta <_{ad} U_i$, we obtain a contradiction for Equation 10.7.
- $(\alpha - \beta) \in \mathsf{altclos}_j(TA_0, TA_{l,i})$, $j > 1$. Then $\exists \gamma : (\beta - \gamma) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. As $Var(\beta) \notin \mathcal{H}$, then $(\beta - \gamma) \in TA_{l,i}$, hence $(\beta - \gamma) \leq_a A_{l,i}$. Knowing this, we obtain $(\gamma - \alpha) \in \mathsf{altclos}_{(j-1)\rightarrow}(TA_0, TA_{l,i})$, which means that $Var(\gamma) \in \mathcal{H}$. With $\beta <_{ad} U_i$, β and γ sharing a common structure, and $Var(\gamma) \in \mathcal{H}$, this implies that $\gamma <_{ad} \overline{U_i}$. With $Var(\alpha) \in \mathcal{H}$ and $Var(\gamma) \in \mathcal{H}$, we can do a similar reasoning as in Item 2.2.1., yet where $\gamma \in \overline{U_i}$, $Var(\gamma) \in \mathcal{H}$ and $Var(\alpha) \in \mathcal{H}$, hence obtaining a contradiction with Equation (10.8).

2.2.4. $Var(\beta) \notin \mathcal{H}, Var(\alpha) \notin \mathcal{H}$.
Again, we have that $(\alpha - \beta)$ is either formed by paths of length 1, *i.e.*, $(\alpha - \beta) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$ or it is formed by paths of length larger than 1, *i.e.*, $(\alpha - \beta) \in \mathsf{altclos}_j(TA_0, TA_{l,i})$, $j > 1$.

- $(\alpha - \beta) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. As neither α nor β relate to head variables, we have $(\alpha - \beta) \in TA_{l,i}$. With $\beta \in U_i$ this contradicts Equation (10.7).
- $(\alpha - \beta) \in \mathsf{altclos}_j(TA_0, TA_{l,i})$, $j > 1$. This means that there exists a $\gamma : (\alpha - \gamma) \in \mathsf{altclos}_1(TA_0, TA_{l,i})$. Knowing that $Var(\alpha) \notin \mathcal{H}$, then $(\alpha - \gamma) \in TA_{l,i}$, and thus $(\gamma - \beta) \in \mathsf{altclos}_{(j-1)\rightarrow}(TA_0, TA_{l,i})$. This means that there must ex-

ist a γ' such that $(\gamma - \gamma') \in \mathsf{altclos}_{1\rightarrow}(TA_0, TA_{l,i}) = TA_0$. With $Vars(TA_0) \subseteq \mathcal{H}$ we have $Var(\gamma) \in \mathcal{H}$. Hence, with $\beta <_{ad} U_i$ and $(\gamma - \beta) \in \mathsf{altclos}_{(j-1)\rightarrow}(TA_0, TA_{l,i})$, we obtain that $\gamma <_{ad} \overline{U_i}$. We can again apply a similar reasoning as in Item 2.2.3, and show that this situation contradicts Equation (10.8).

The case studies can be repeated for all data structures subsumed by D_i, hence proving the theorem.

□

In the case of a possibility of direct reuse, the above theorem states that if the goal-independent analysis detects that a data structure α can be reused at a program point (i) knowing that U_i is the set of data structures that is certainly live, and $A_{l,i}$ is the set of pairs of data structures known to be sharing memory, then verifying the reuse for a call description in general using $R_i = \langle\!\langle \{\alpha\}, U_i, A_{l,i} \rangle\!\rangle$ can be replaced by verifying reuse w.r.t. the compacted reuse information, *i.e.*, $\overline{R_i} = \mathsf{compact}(R_i, \mathcal{H})$. Therefore, instead of storing R_i in an optimisation interface file, it suffices to store $\overline{R_i}$. As $Vars(\overline{R_i}) \subseteq \mathcal{H}$, where $\mathcal{H}$ is the set of head variables of the procedure in which the deconstruction unification for the direct reuse occurs, we have a form of guarantee that $\overline{R_i}$ will in general be more compact to represent, and more efficient to compute with than R_i.

We illustrate the above ideas with the example of append/3, the concatenation of lists.

Example 10.1 *Based on the code of* append *given at page 212 we obtain at program point* (3) *that* $U_3 = \{Xe^{\overline{e}}, Xs^{\overline{e}}, Y^{\overline{e}}\}$ *are the data structures in use at that program point, yet there are no local structure sharing pairs — hence* $A_{l,3} = \{\,\}$*, and* $X^{\overline{e}}$ *is the candidate data structure for being reused — thus* $D_3 = \{X^{\overline{e}}\}$*. The reuse information for that program point is therefore:*

$$R_3 = \langle\!\langle D_3, U_3, A_{l,3} \rangle\!\rangle = \langle\!\langle \{X^{\overline{e}}\}, \{Xe^{\overline{e}}, Xs^{\overline{e}}, Y^{\overline{e}}\}, \{\,\} \rangle\!\rangle$$

The compacted reuse information is

$$\begin{array}{rcl} \overline{R_3} & = & \langle\!\langle \overline{D_3}, \overline{U_3}, \overline{A_{l,3}} \rangle\!\rangle \\ & = & \left\langle\!\!\left\langle \begin{array}{l} \mathsf{compact}(D_3, \{X, Y, Z\}), \mathsf{compact}(U_3, \{X, Y, Z\}), \\ \qquad\qquad\qquad\qquad \mathsf{compact}(A_{l,3}, \{X, Y, Z\}) \end{array} \right\rangle\!\!\right\rangle \\ & = & \langle\!\langle \{X^{\overline{e}}\}, \{Y^{\overline{e}}\}, \{\,\} \rangle\!\rangle \end{array}$$

We study a number of different call descriptions:

1. $AL = \langle \{\,\}, \{\,\} \rangle$*. This is the empty call description, hence reuse will of course be safe.*

2. $\langle A_0, L_0\rangle = \langle\{\left(X^{\overline{([],1)}} - X^{\overline{([],1)}}\right)\}, \{Z^{\overline{e}}\}\rangle$. *This means that only the elements of X can be shared, and except for $Z^{\overline{e}}$ there are no other live data structures. In this situation*

$$\begin{aligned}
&\mathsf{extend}_a(L_0, \overline{A_{l,3}}) \sqcup_{ad} \mathsf{extend}_a(\overline{U_3}, \mathsf{comb}_a(A_0, \overline{A_{l,3}})) \\
&\quad = \quad \mathsf{extend}_a(\{Z^{\overline{e}}\}, \{\,\}) \\
&\qquad\qquad \sqcup_{ad} \mathsf{extend}_a(\{Y^{\overline{e}}\}, \{\left(X^{\overline{([],1)}} - X^{\overline{([],1)}}\right)\}) \\
&\quad = \quad \{Y^{\overline{e}}, Z^{\overline{e}}\}
\end{aligned}$$

With $\overline{D_3} = \{X^{\overline{e}}\}$ and $X^{\overline{e}} \not<_{ad} \{Y^{\overline{e}}, Z^{\overline{e}}\}$, reuse is safe. Hence, even if the elements of the list that is going to be reused are shared (or live, one can repeat the exercise), reuse is still confirmed by checking the (compacted) reuse information.

3. $\langle A_0, L_0\rangle = \langle\{\left(X^{\overline{e}} - Y^{\overline{e}}\right)\}, \{Z^{\overline{e}}\}\rangle$. *In this case,*

$$\begin{aligned}
&\mathsf{extend}_a(L_0, \overline{A_{l,3}}) \sqcup_{ad} \mathsf{extend}_a(\overline{U_3}, \mathsf{comb}_a(A_0, \overline{A_{l,3}})) \\
&\quad = \quad \mathsf{extend}_a(\{Z^{\overline{e}}\}, \{\,\}) \sqcup_{ad} \mathsf{extend}_a(\{Y^{\overline{e}}\}, \{\left(X^{\overline{e}} - Y^{\overline{e}}\right)\}) \\
&\quad = \quad \{X^{\overline{e}}, Y^{\overline{e}}, Z^{\overline{e}}\}
\end{aligned}$$

In this case $X^{\overline{e}} <_{ad} \overline{D_3}$, yet $X^{\overline{e}} <_{ad} \{X^{\overline{e}}, Y^{\overline{e}}, Z^{\overline{e}}\}$, hence reuse is not allowed because indeed, due to the fact that X and Y are shared in memory, and Y is live, the list-cells of X may not be reused.

There is an interesting case when in the compacted reuse information the set of data structures referring to the reusable heap cells is the empty set. This is illustrated by the following example:

Example 10.2 *Consider the following program:*

```
%:- type thingy ---> thing(int,int).
%:- pred generate(thingy).
%:- mode generate(out) is det.
generate(T) :- T = thing(0,0).

%:- pred compute(thingy).
%:- mode compute(out) is det.
compute(T) :-
    (1) generate(X),
    (2) X => thing(A,B),
    (3) A1 = A + 1,
    (4) B1 = B + 1,
    (5) T <= thing(A1,B1).
```

In compute/1*, X is a local variable that is instantiated in* generate/1*. It is then deconstructed, and not used elsewhere, hence it should be detected as a potential candidate for*

reuse. In the presence of the construction of a new variable T we have a matching deconstruction/construction pair, hence a possibility of direct reuse. The reuse information at program point (2) *is* $R_2 = \langle\!\langle \{X^{\overline{\epsilon}}\}, \{A^{\overline{\epsilon}}, B^{\overline{\epsilon}}\}, \{\,\} \rangle\!\rangle$. *The compacted form of this information is* $\overline{R_2} = \mathsf{compact}(R_2, \{T\}) = \langle\!\langle \{\,\}, \{\,\}, \{\,\} \rangle\!\rangle$. *The fact that the first component is empty signals that the heap cells that can be reused in* compute/1 *have no connection with the head variables, hence, the liveness nor the structure sharing information when calling* compute/1 *can ever make these heap cells live. This means that the heap cells used by* $X^{\overline{\epsilon}}$ *can always be reused, no matter what the call description of* compute/1 *is.*

This phenomenon is also covered by Theorem 10.1, in the sense that if $\overline{D_i} = \mathsf{compact}(D_i, \mathcal{H}) = \{\,\}$ — where $\mathcal{H}$ represents the set of head variables as usual, then there can not exist any $\beta <_{ad} \overline{D_i}$ such that β may not be subsumed by the result of extending the live and in use data structures using the structure sharing information. We formalise this in the following corollary.

Corollary 10.2 *Let there be a possibility of reuse in a procedure q at program point* (i). *Let* $R_i = \langle\!\langle D_i, U_i, A_{l,i} \rangle\!\rangle$ *be the corresponding reuse information. The compacted reuse information w.r.t. the set of head variables* $\mathcal{H}$ *is*

$$\overline{R_i} = \langle\!\langle \mathsf{compact}(D_i, \mathcal{H}), \mathsf{compact}(U_i, \mathcal{H}), \mathsf{compact}(A_{l,i}, \mathcal{H}) \rangle\!\rangle$$

If $\mathsf{compact}(D_i, \mathcal{H}) = \{\,\}$, *then for every call description* AL *of* q, *we have* $AL \sharp \overline{R_i}$, *hence the reuse at program point* (i) *is always safe.*

Reuse opportunities where the compacted reuse information has no references to the reusable cells are called *unconditional reuses*. In general, we have:

Definition 10.5 (Unconditional Reuse) *A reuse opportunity with reuse information* R_i *is called an* unconditional reuse *if for all call descriptions* AL, $AL \sharp R_i$ *holds.*

This is especially the case for all reuses with compacted reuse information having their compacted set of reusable data structures reduced to the empty set. Thus, with $\mathcal{H}$ the set of head variables of the procedure to which the reuse information tuple belongs, we have:

$$\mathsf{compact}(R_i, \mathcal{H}) = \langle\!\langle \mathsf{compact}(D_i, \mathcal{H}), \mathsf{compact}(U_i, \mathcal{H}), \mathsf{compact}(A_{l,i}, \mathcal{H}) \rangle\!\rangle$$

where $\mathsf{compact}(D_i, \mathcal{H}) = \{\,\}$.

Reuses that are not unconditional are called *conditional*.

Unconditional reuses are always safe. This means that in our version generation strategy we can safely decide to create two versions of each procedure: one version that contains only the unconditional reuses, if at all, and one version that realises all of the detected reuses for the goal-independent analysis situation: the conditional as well as the unconditional ones. If these two versions are identical, of course only one version needs to be created.

10.3.5 Reuse Information: Indirect Reuse

We now investigate what the reuse information consists of for cases of indirect reuse.

Consider the following code fragment:

```
p(P1,...,Pm) :- ..., (i) q(X1,...,Xn), ... .
q(Q1,...,Qn) :- ..., (j) Y => f(...), ... .
```

Let R_q be the compacted reuse information for the heap cells that may become available for reuse at program point (j) in procedure q/n. Let $AL_{p,\delta}$ be the empty call description for p/m, and $AL_{i,\delta}$ the resulting description at program point (i). Upon projection and renaming, we can verify whether a call to the reuse version of q/n is safe in the goal-independent setting. Suppose $\rho_{\overline{X}\to\overline{Q}}\left((AL_{i,\delta})|_{\overline{X}}\right) \sharp R_q$ which means that a call to the reuse version of q is safe for the empty call description of p/m, then we can generate a version for p/m that explicitly calls the reuse version of q/n. Now we need to derive the reuse information for that indirect reuse so that the safeness of calls to the reuse version of p/m can be verified. The task of deriving the reuse information for indirect reuse is the subject of this section.

The intuition is as follows. Independent of the actual call description of p/m, we always have the local structure sharing set $A_{l,i}$ and a set of data structures that is live at that program point, namely U_i, the set of data structures in forward and backward use w.r.t. program point (i). On the other hand we know from the reuse information $R_q = \langle\!\langle D_q, U_q, A_q \rangle\!\rangle$ of q that the heap cells pointed at by D_q may be reused if they remain dead taking into account the extra constraints from the call description, and the fact that U_q and A_q are the live structures and aliases present at the program point where the reuse is decided. We can bring this information to the context of p by renaming all the involved variables (which are limited to the head variables anyway), and taking into account the local new constraints. Hence, we obtain the new reuse information:

$$R_i = \left\langle\!\!\left\langle \rho_{\overline{Q}\to\overline{X}}\left(D_q\right) \text{ , } \rho_{\overline{Q}\to\overline{X}}\left(U_q\right) \sqcup_{ad} U_i \text{ , } \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)) \right\rangle\!\!\right\rangle$$

We will show that we can safely use R_i as the reuse information for verifying the call to the reuse version of q. Note that just as for direct reuses, we can compact this information to the head variables, in this case $\{P_1, \ldots, P_m\}$, hence only recording $\overline{R_i} = \mathsf{compact}(R_i, \{P_1, \ldots, P_m\})$ in the interface file of the module in which p/m is defined.

Before proving the above presented ideas, we introduce two lemma's that are of use in that proof. The first lemma states that extending a given set subsequently by two different structure sharing sets can be approximated by extending that set with the combination of these two sharing sets. The second lemma shows the associativity property of comb_a.

Lemma 10.3 *For every set of abstract data structures L, and for every sets of abstract sharing data structures A_1 and A_2 we have:*

$$\mathsf{extend}_a(\mathsf{extend}_a(L, A_1), A_2) \sqsubseteq_a \mathsf{extend}_a(L, \mathsf{comb}_a(A_1, A_2))$$

Proof The set of abstract data structures $\mathsf{extend}_a(L, A_1)$ is formed by the elements of L, and the edges that can be reached by constructing paths of length 1 from an element from L using paths subsumed by A_1. This means that elements of $\mathsf{extend}_a(\mathsf{extend}_a(L, A_1), A_2)$ are elements of $\mathsf{extend}_a(L, A_1)$, and the elements that can be reached by constructing paths of length 1 from elements of that set, using paths subsumed by A_2, which is the same as saying that this set consists of elements from L, elements that are reached by constructing paths of length 1 using vertices from A_1 or A_2, and elements that are reached by constructing paths of length 2, starting from elements of L, with a first vertex from A_1 and a second vertex from A_2. In terms of the notion of alternating closure, we have:

$$\begin{array}{l} \mathsf{extend}_a(\mathsf{extend}_a(L, A_1), A_2) \\ \quad = \quad \mathsf{extend}_a(L, A_1 \sqcup_a A_2 \sqcup_a \mathsf{altclos}_{2\rightarrow}(A_1, A_2)) \end{array}$$

Knowing that $\mathsf{altclos}(A, B)$ can be rewritten to

$$\begin{array}{rcl} \mathsf{altclos}(A, B) & = & \mathsf{altclos}_1(A, B) \cup \mathsf{altclos}_2(A, B) \cup \ldots \\ & = & A \cup B \cup altclos_2(A, B) \cup \ldots \end{array}$$

we can decompose $\mathsf{comb}_a(A_1, A_2)$ to

$$\begin{array}{rcl} \mathsf{comb}_a(A_1, A_2) & = & \mathsf{termshift}(A_1) \cup \mathsf{termshift}(A_2) \\ & & \cup \mathsf{altclos}_2(\mathsf{termshift}(A_1) \cup \mathsf{termshift}(A_2)) \cup \ldots \end{array}$$

This means that $A_1 \sqcup_a A_2 \sqcup_a \mathsf{altclos}_{2\rightarrow}(A_1, A_2) \sqsubseteq_a \mathsf{comb}_a(A_1, A_2)$, hence proving the lemma.

□

We now generalise the above presented ideas in the following theorem:

Theorem 10.2 (Indirect Reuse Information) *Consider the following setting:*

p(P1,...,Pm) :- ..., (i) q(X1,...,Xn), ...
q(Q1,...,Qn) :- ..., (j) ..., ...

Let $R_q = \langle\!\langle D_q, U_q, A_q \rangle\!\rangle$ be the (compacted) reuse information concerning a potential reuse at program point (j) in the definition of procedure q/n. Let U_i and $A_{l,i}$ be the set of data structures in forward/backward use, and the local set of structure sharing pairs respectively. Let $R_i = \langle\!\langle D, U, A_l \rangle\!\rangle$, where $D = \rho_{\overline{Q}\rightarrow\overline{X}}(D_q)$, $U = \rho_{\overline{Q}\rightarrow\overline{X}}(U_q) \sqcup_{ad} U_i$, and $A_l = \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\rightarrow\overline{X}}(A_q))$.

Let AL be a call description for p/m, and AL_i the corresponding description at program point i, then:

$$AL \sharp R_i \Rightarrow \rho_{\overline{X}\to\overline{Q}}\left((AL_i)|_{\overline{X}}\right) \sharp R_q$$

Proof Consider a call description $AL = \langle A_0, L_0\rangle$ and let $A_{l,i}$ be the local structure sharing component at program point (i) in the procedure definition of p/m. Then AL_i, the description obtained at program point (i) is given by $AL_i = \langle A_i, L_i\rangle$ where $A_i = \mathsf{comb}_a(A_0, A_{l,i})$ and $L_i = \mathsf{extend}_a(L_0, A_{l,i}) \cup \mathsf{extend}_a(U_i, A_i)$, *i.e.*, $L_i = \mathsf{extend}_a(L_0, A_{l,i}) \cup \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})$.

We expand the proposition $AL\sharp R_i$ to its full length, namely that:

$$\begin{array}{l}\forall \alpha <_{ad} \rho_{\overline{Q}\to\overline{X}}\left(D_q\right) : \\ \quad \alpha \not<_{ad} \quad \mathsf{extend}_a(L_0, \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)))\sqcup_{ad} \\ \qquad\qquad \mathsf{extend}_a(\rho_{\overline{Q}\to\overline{X}}\left(U_q\right) \sqcup_{ad} U_i, \mathsf{comb}_a(A_0, \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right))))\end{array}$$

which is equivalent to:

$$\begin{array}{lll}\forall \alpha <_{ad} \rho_{\overline{Q}\to\overline{X}}\left(D_q\right) : & & \\ \quad \alpha \not<_{ad} & \mathsf{extend}_a(L_0, \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right))) & (a) \\ \wedge \quad \alpha \not<_{ad} & \mathsf{extend}_a(\rho_{\overline{Q}\to\overline{X}}\left(U_q\right), \mathsf{comb}_a(A_0, \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)))) & (b) \\ \wedge \quad \alpha \not<_{ad} & \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, \mathsf{comb}_a(A_{l,i}, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)))) & (c)\end{array}$$

Similarly $\rho_{\overline{X}\to\overline{Q}}\left((AL_i)|_{\overline{X}}\right) \sharp R_q$ which is equivalent to $(AL_i)|_{\overline{X}} \sharp \rho_{\overline{Q}\to\overline{X}}\left(R_q\right)$ can be expanded to:

$$\begin{array}{ll}\forall \alpha <_{ad} \rho_{\overline{Q}\to\overline{X}}\left(D_q\right) : & \\ \alpha \not<_{ad} & \\ \quad \mathsf{extend}_a(\,(\mathsf{extend}_a(L_0, A_{l,i}) \sqcup_{ad} \mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})))|_{\overline{X}}\,, & \\ \quad \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)) & (1) \\ \quad \sqcup_{ad}\mathsf{extend}_a(\rho_{\overline{Q}\to\overline{X}}\left(U_q\right), \mathsf{comb}_a(\mathsf{comb}_a(A_0, A_{l,i}), \rho_{\overline{Q}\to\overline{X}}\left(A_q\right))) & (2)\end{array}$$

This can be rewritten in the same way as done for $AL\sharp R_i$, yielding:

$$\begin{array}{lll}\forall \alpha <_{ad} \rho_{\overline{Q}\to\overline{X}}\left(D_q\right) : & & \\ \quad \alpha \not<_{ad} & \mathsf{extend}_a(\,(\mathsf{extend}_a(L_0, A_{l,i}))|_{\overline{X}}\,, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)) & (A) \\ \wedge \quad \alpha \not<_{ad} & \mathsf{extend}_a(\,(\mathsf{extend}_a(U_i, \mathsf{comb}_a(A_0, A_{l,i})))|_{\overline{X}}\,, \rho_{\overline{Q}\to\overline{X}}\left(A_q\right)) & (B) \\ \wedge \quad \alpha \not<_{ad} & \mathsf{extend}_a(\rho_{\overline{Q}\to\overline{X}}\left(U_q\right), \mathsf{comb}_a(\mathsf{comb}_a(A_0, A_{l,i}), \rho_{\overline{Q}\to\overline{X}}\left(A_q\right))) & (C)\end{array}$$

Terms (A) and (B) stem from term (1) in the previous expression by distributing extend_a over the two sets of data structures.

Using Lemma 10.3 we can show that the expression in (A) is subsumed by the expression in (a). A combination of Lemma 10.3 and the fact that comb_a

is associative (c.f. Page 134) allows us to conclude that (B) is approximated by (c), and due to the associative nature of comb_a we have that the expression in (C) is subsumed by (b). Hence, if $AL \sharp R_i$ then $\rho_{\overline{X}\to\overline{Q}}\left((AL_i)|_{\overline{X}}\right)\sharp R_q$. □

Example 10.3 *For the code of* append *given at page 212 we obtained the compacted reuse information for the deconstruction at program point* (3)

$$\overline{R_3} = \mathsf{compact}(R_3, \{X, Y, Z\}) = \langle\!\langle \{X^{\overline{\epsilon}}\}, \{Y^{\overline{\epsilon}}\}, \{\,\} \rangle\!\rangle$$

(Example 10.1). Note that $\overline{R_3}$ *corresponds to the reuse information* R_q *as used in the previous theorem.*

For the empty call description, the liveness description at program point (4)*, the recursive call, is:*

$$AL_4 = \langle \{ \left(X^{\overline{([],1)}} - Xe^{\overline{\epsilon}} \right), \left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}} \right) \}, \{ Xe^{\overline{\epsilon}}, X^{\overline{([],1)}}, Zs^{\overline{\epsilon}} \} \rangle$$

Projecting AL_4 *on the actual variables of the recursive call to* append *and renaming the result to the formal head variables we obtain the call description*

$$AL = \rho\left((AL_4)|_{\{Xs,Y,Zs\}} \right) = \rho\left(\langle \{\,\}, \{Zs^{\overline{\epsilon}}\} \rangle \right) = \langle \{\,\}, \{Z^{\overline{\epsilon}}\} \rangle$$

where ρ *is the renaming function mapping the actual variables* $\{Xs, Y, Zs\}$ *onto the formal head variables* $\{X, Y, Z\}$*. Clearly* $AL \sharp \overline{R_3}$*, which means that under the empty call description, the recursive call to append also allows the reuse of the list backbone of the first input list. This indirect reuse introduces its own reuse information: the reusable heap cells are the renaming of the reusable heap cells in* $\overline{R_3}$*, the in use information is the in use information of* $\overline{R_3}$ *updated with the in use information at program point* (4) *(where* $U_4 = \{Xe^{\overline{\epsilon}}, Zs^{\overline{\epsilon}}\}$*), and finally, the structure sharing part is the combination of the structure sharing recorded in* $\overline{R_3}$ *and the structure sharing available at* (4)*. Thus,*

$$R_4 = \left\langle\!\!\left\langle \{Xs^{\overline{\epsilon}}\}, \{Y^{\overline{\epsilon}}, Xe^{\overline{\epsilon}}, Zs^{\overline{\epsilon}}\}, \{ \left(X^{\overline{([],1)}} - Xe^{\overline{\epsilon}} \right), \left(X^{\overline{\epsilon}} - Xs^{\overline{\epsilon}} \right) \} \right\rangle\!\!\right\rangle$$

Just as for the direct reuse R_3*, we can translate the reuse information* R_4 *to the head variables of* append *by compacting it. We obtain:*

$$\overline{R_4} = \mathsf{compact}(R_4, \{X, Y, Z\}) == \langle\!\langle \{X^{\overline{\epsilon}}\}, \{Y^{\overline{\epsilon}}\}, \{\,\} \rangle\!\rangle$$

Here $\overline{R_4} = \overline{R_3}$ *which means that if a call description* AL *satisfies* $\overline{R_3}$ *— the first list cell of the first input list can be reused, then it clearly also satisfies* $\overline{R_4}$*. Hence, all list cells of the first list can be reused.*

The above example illustrates the need for a fixpoint computation in case of recursive procedure definitions. Indeed, if in the above example $\overline{R_4}$ would not

have been equal to $\overline{R_3}$, then a new iteration would have been needed to verify whether the projected and renamed liveness description AL_4 also verifies the reuse information $\overline{R_4}$. If so, then that information would have to be updated with the local information at (4) and compacted again. This process needs to be repeated until a fixpoint is reached.

10.4 Putting it all together

We have shown that for deriving liveness information for the program points within a procedure it suffices to know the liveness call description and the local structure sharing information built by each of the called procedures. When deriving reuse information, we have decided to detect all possibilities of reuse within a procedure by assuming that the liveness call description is the empty call description. Hence, when putting these two considerations together, we obtain a goal-independent analysis deriving sets of reuse information based on goal-independent liveness information of the program.

To formalise these processes we need to formalise the set of reuse information tuples as a lattice domain.

Definition 10.6 (Reuse Information Domain $\mathcal{RI}$) *The domain of reuse information tuples consists of $\wp(\langle\!\langle \wp(\overline{\mathcal{D}}_{VI}), \wp(\overline{\mathcal{D}}_{VI}), \wp(\overline{\mathcal{SD}}_{VI}) \rangle\!\rangle)$, which we abbreviate to $\mathcal{RI}$.*

Each individual reuse information tuple R describes a set of call descriptions $A = \{AL_1, AL_2, \ldots, AL_n\}$ for which the relation $AL_i \sharp\ R, 1 \leq i \leq n$ holds. We can show that this relation is monotonic in its first argument. This means that $\forall AL_1, AL_2 \in \mathcal{AL}$ and $AL_1 \sqsubseteq_{al} AL_2$, if $AL_2 \sharp\ R$, then also $AL_1 \sharp\ R$. The relation is also additive: if $AL_1 \sharp\ R$ and $AL_2 \sharp\ R$, then $(AL_1 \sqcup_{al} AL_2) \sharp\ R$. As the domain of abstract liveness descriptions is a complete lattice this implies that the least upper bound of all liveness descriptions satisfying a particular reuse information tuple, exists and also satisfies that same tuple. Thus, let $AL = \sqcup_{al}\{AL' \mid AL' \sharp\ R\}$, for $R \in \mathcal{RI}$, then $AL \sharp\ R$. This least upper bound is unique, which means that each reuse information tuple can be uniquely mapped onto one single call description describing all the calls for which the reuse described by the tuple is safe. We call this abstract liveness call description the *associated call description* of a reuse tuple.

Definition 10.7 (Associated call description AL_R) *Let $R \in \mathcal{RI}$ be a reuse information tuple, then the* associated call description *of R, denoted by AL_R is the least upper bound of all the call descriptions that satisfy the reuse tuple. Formally: $AL_R = \sqcup_{al}\{AL \mid AL \sharp R\}$.*

Given this one-to-one possibility of mapping reuse information tuples onto abstract liveness descriptions, the ordering in $\mathcal{RI}$ becomes equivalent to the ordering in $\mathcal{AL}$. Therefore, expressions such as $AL \sharp\ R$ and $AL \sqsubseteq_{al} AL_R$ are equival-

ent. We lift the applicability of the $\sharp$-operation, and redefine it as an order-relation in $\mathcal{RI}$.

Definition 10.8 (Ordering of Reuse Information Tuples) *Let R_1, R_2 be two individual reuse information tuples, then R_1 is subsumed by R_2, denoted by $R_2 \sharp R_1$, iff $AL_{R_2} \sharp R_1$, which is equivalent to $AL_{R_2} \sqsubseteq_{al} AL_{R_1}$, where AL_{R_1} and AL_{R_2} are the associated call descriptions to R_1, resp. R_2.*

In practice, the above definition will not really be usable as it theoretically requires the associated call descriptions of the reuse information tuples, which are not trivial to determine.

A more practical order relation is inspired by the following observation:

Corollary 10.3 *Let $R_1 = \langle\!\langle D_1, U_1, A_1 \rangle\!\rangle$ and $R_2 = \langle\!\langle D_2, U_2, A_2 \rangle\!\rangle$ be two reuse information tuples. Then*

$$(D_1 = \{\,\} \text{ or } \begin{pmatrix} D_1 \sqsubseteq_{ad} D_2 \\ U_1 \sqsubseteq_{ad} U_2 \\ A_1 \sqsubseteq_{a} A_2 \end{pmatrix}) \quad \Rightarrow \quad R_2 \sharp R_1$$

Note that the case of $D_1 = \{\,\}$ reflects unconditional reuse, hence is subsumed by any other reuse information tuple.

We use the following ordering between reuse information tuples:

Definition 10.9 *Let $R_1 = \langle\!\langle D_1, U_1, A_1 \rangle\!\rangle$ and $R_2 = \langle\!\langle D_2, U_2, A_2 \rangle\!\rangle$ be two reuse information tuples, then R_1 is subsumed by R_2, denoted by $R1 \ll_r R_2$, iff $D_1 = \{\,\}$ or $(D_1 \sqsubseteq_{ad} D_2 \wedge U_1 \sqsubseteq_{ad} U_2 \wedge A_1 \sqsubseteq_{a} A_2$.*

Obviously, two reuse information tuples are considered to be equivalent if they are mutually subsumed. Hence, let $R_1, R_2 \in \mathcal{RI}$, then $R_1 \equiv R_2$ iff $R_1 \ll_r R_2$ and $R_2 \ll_r R_1$.

The domain of sets of reuse information tuples is then ordered by the set-inclusion operation, modulo equivalence of the reuse information tuples.

Definition 10.10 (Ordering in $\mathcal{RI}$) *Let R be a reuse information tuple, and $RI \in \mathcal{RI}$, then R is subsumed by RI, which is denoted by $R \leq_r RI$, iff $\exists R' \in RI$, such that $R' \equiv R$.*

If $RI_1, RI_2 \in \mathcal{RI}$, then RI_1 is subsumed by RI_2 iff $\forall R \in RI_1 : R \leq_r RI_2$. This is denoted by $R_1 \sqsubseteq_r R_2$.

And finally, the least upper bound of two sets of reuse information tuples is simply defined as their union: $RI_1 \sqcup_r RI_2 = RI_1 \cup RI_2$, where $\cup$ is the set-union operation.

This domain is a lattice with bottom element the empty set, and top element the set of all possible combinations of reuse information tuples over the given set of variables.

In the previous sections we presented how reuse information can be gathered using goal-independent liveness information of the procedures. We have seen that this information does not require an explicit analysis to be set up as liveness information at a given program point can always be computed using forward use information, backward use information, and the local component for structure sharing information. This means that by assuming these basic types of information available, we can immediately express the liveness descriptions, hence derive the reuse information tuples.

We therefore assume that forward use, backward use and structure sharing as well are ready available in the program. Forward use can be queried using the operation forward, backward use is given with the operation backward. Similarly, we introduce the implicit operation sharing that, given a specific program point, returns the local structure sharing component for that program point. Formally, if $\mathbf{R}_{M\star p}^{\wp(\overline{\mathcal{SD}}_{VI})}$ is the semantic function of a rulebase in the context of the pre-annotating goal-independent semantics $\mathcal{S}em_{M\star p}$ instantiated for the domain $\wp(\overline{\mathcal{SD}}_{VI})$, with the auxiliary operations given in Definition 6.32, then $\mathsf{sharing} : \mathsf{pp} \to \wp(\overline{\mathcal{SD}}_{VI})$ is defined as:

$$\mathsf{sharing}(i) \quad = \quad \begin{array}{l}\text{let } A = \mathbf{R}_{M\star p}^{\wp(\overline{\mathcal{SD}}_{VI})}[\![r]\!] \text{ in} \\ \quad A(i)\end{array}$$

Note that gathering this goal-independent rule-base meaning in the presence of modules requires that the structure sharing of procedure defined in other modules than the one actually analysed be recorded in an adequate interface file as was described in Section 10.2.

We can now define the actual derivation of reuse information.

Definition 10.11 (Reuse information derivation) *The derivation of reuse information is defined as the goal-independent semantic functions $\mathcal{S}em_{M\bullet}$ instantiated with the domain $\mathcal{RI}$ and where* init_r *is defined as* $\mathsf{init}_r = \{\,\}$*, adding the result of a unification is defined as*

$$\begin{array}{lcl}
\mathsf{add}_r((X := Y), RI) & = & RI \\
\mathsf{add}_r((X <= f(\overline{Y})), RI) & = & RI \\
\mathsf{add}_r((X == Y), RI) & = & RI \\
\mathsf{add}_r((X => f(\overline{Y}), RI) & = & \\
\quad \textit{let } i & = & \mathsf{pp}(X => f(\overline{Y})) \textit{ in} \\
\quad \textit{let } A_{l,i} & = & \mathsf{sharing}(i) \textit{ in} \\
\quad \textit{let } U_i & = & \mathsf{data}_a(\mathsf{forward}(i) \cup \mathsf{backward}(i)) \textit{ in} \\
\quad \textit{let } R_i & = & \langle\!\langle \{X^{\overline{e}}\}, U_i, A_{l,i} \rangle\!\rangle \textit{ in} \\
\quad \textit{let } AL_\delta & = & \langle \{\,\}, \{\,\} \rangle \textit{ in} \\
\quad \textit{if } AL_\delta \sharp R_i \textit{ then } \{\mathsf{compact}(R_i, \mathcal{H})\} \sqcup_r RI \textit{ else } RI
\end{array}$$

where $\mathcal{H}$ represent the head variables of the procedure to which the literal belongs. Combining the results of a procedure call with the current reuse information is given by:

$$
\begin{array}{lcl}
\mathsf{comb}_r(p(\overline{X}), RI_o, RI_n) & = & \\
\quad \textit{let } i & = & \mathsf{pp}(p(\overline{X}))\ \textit{in} \\
\quad \textit{let } A_{l,i} & = & \mathsf{sharing}(i)\ \textit{in} \\
\quad \textit{let } U_i & = & \mathsf{data}_a(\mathsf{forward}(i) \cup \mathsf{backward}(i))\ \textit{in} \\
\quad \textit{let } RI'_n & = & \left\{ \langle\!\langle D, U', A' \rangle\!\rangle \;\middle|\; \begin{array}{l} \langle\!\langle D, U, A \rangle\!\rangle \in RI_n, \\ U' = U \sqcup_{ad} U_i, \\ A' = \mathsf{comb}_a(A_{l,i}, A) \end{array} \right\}\ \textit{in} \\
\quad \textit{let } AL_\delta & = & \langle \{\,\}, \{\,\} \rangle\ \textit{in} \\
& & \textit{if } \forall R' \in RI'_n : AL_\delta \sharp R' \\
& & \textit{then } RI_o \sqcup_r \{\mathsf{compact}(R', \mathcal{H}) | R' \in RI'_n\} \\
& & \textit{else } RI_o
\end{array}
$$

where again $\mathcal{H}$ represents the set of head variables of the procedure to which the procedure call belongs.

Note that in the previous definition we have slightly adapted the signature of the combination function in the sense that we have added the extra argument of the actual called procedure. This is only needed to know the program point of that literal. Observe that the definitions of add_r and comb_r are very similar.

10.5 Prototype Implementation

We adapted the prototype described in Chapter 9 to allow the use of modules in a program. This prototype was developed with elder insights in mind, and therefore presents some differences with the above presented theory. We first present these differences, then detail some of the implementation issues, and finally present and discuss the benchmarks run with that prototype.

10.5.1 Liveness Definition

In Section 8.5 we have shown how we can increase the precision of the set of live data structures by separating the liveness of the call description from the new live structures in a procedure, in the definition of live and therefore also in live_a. We call live and live_a as defined by Equation (8.1), resp. Equation (8.3), the *old* liveness definition, while Equation (8.5) and Equation (8.6) define the *new* liveness functions.

The increase in precision of the new live_a definition w.r.t. the old definition is especially apparent in the goal-dependent semantics where due to the non-idempotence of the old live_a, the liveness of the call description would be extended twice w.r.t. the structure sharing of that call description. Yet, in the context

of a goal-independent analysis of liveness information, this issue becomes less important. Indeed, we can show that the abstract liveness information derived using the goal-independent liveness descriptions is inherently more precise than when derived in a pure goal-dependent derivation.

In this prototype, we still used the old liveness definitions, and although this approach is know to be less precise, we have not investigated the loss of precision further.

10.5.2 Default Liveness Analysis

In the process of modularising the reuse analysis, we have decided that only two versions of each procedure will at most be created: one version that implements all possible discovered reuse, and one version that only implements the unconditional reuses, if at all. All the possible reuses are discovered by assuming an empty liveness call description.

However, this approach may find too many opportunities of reuse as it may be unlikely that an actual call will ever have its liveness component empty. At least the output variables of the procedure are in most cases live in the calling environment of the procedure. Therefore, in order to detect the possibilities of reuse within a procedure p we assume that its output variables are live. We call this the *default* liveness information of p. Therefore, instead of deriving reuse information tuples based on a goal-independent liveness analysis, we have implemented our prototype so that reuse information is derived based on the default liveness descriptions of the procedures considered.

Formally this requires a slight adaptation of the auxiliary operations add_r and comb_r, and also an adaptation of the underlying semantics in the sense that the output variables of the analysed procedure must be threaded through the semantic functions. As an effect, add_r and comb_r both have an extra argument. That extra argument can either be the analysed procedure itself, or only its output arguments. For clarity of the formalisation, we assume the latter, and obtain the following new definitions:

$$
\begin{array}{l}
\mathsf{add}_r(\mathcal{H}_{\mathtt{out}}, X => f(\overline{Y}), RI) \quad = \\
\quad \text{let } i \quad = \quad \mathsf{pp}(X => f(\overline{Y})) \text{ in} \\
\quad \text{let } A_{l,i} \quad = \quad \mathsf{sharing}(i) \text{ in} \\
\quad \text{let } U_i \quad = \quad \mathsf{data}_a(\mathsf{forward}(i) \cup \mathsf{backward}(i)) \text{ in} \\
\quad \text{let } R_i \quad = \quad \langle\!\langle \{X^{\overline{e}}\}, U_i, A_{l,i} \rangle\!\rangle \text{ in} \\
\quad \text{let } AL_\delta \quad = \quad \langle \{\,\}, \mathsf{data}_a(\mathcal{H}_{\mathtt{out}}) \rangle \text{ in} \\
\quad \text{if } AL_\delta \sharp R_i \text{ then } \{\mathsf{compact}(R_i, \mathcal{H})\} \sqcup_r RI \text{ else } RI
\end{array}
$$

and

$$
\begin{array}{l}
\mathsf{comb}_r(\mathcal{H}_{\mathtt{out}}, p(\overline{X}), RI_o, RI_n) \quad = \\
\quad \text{let } i \quad = \quad \mathsf{pp}(p(\overline{X})) \text{ in} \\
\quad \text{let } A_{l,i} \quad = \quad \mathsf{sharing}(i) \text{ in} \\
\quad \text{let } U_i \quad = \quad \mathsf{data}_a(\mathsf{forward}(i) \cup \mathsf{backward}(i)) \text{ in} \\
\quad \text{let } RI'_n \quad = \quad \left\{ \langle\!\langle D, U', A' \rangle\!\rangle \;\middle|\; \begin{array}{l} \langle\!\langle D, U, A \rangle\!\rangle \in RI_n, \\ U' = U \sqcup_{ad} U_i, \\ A' = \mathsf{comb}_a(A_{l,i}, A) \end{array} \right\} \text{ in} \\
\quad \text{let } AL_\delta \quad = \quad \langle \{\,\}, \mathsf{data}_a(\mathcal{H}_{\mathtt{out}}) \rangle \text{ in} \\
\qquad \text{if } \forall R' \in RI'_n : AL_\delta \sharp R' \\
\qquad \text{then } RI_o \sqcup_r \{\mathsf{compact}(R', \mathcal{H}) | R' \in RI'_n\} \\
\qquad \text{else } RI_o
\end{array}
$$

where $\mathcal{H}$ represents the set of head variables of the procedure to which the literal in question belongs, and $\mathcal{H}_{\mathtt{out}}$ represents the set of output variables of that procedure.

10.5.3 Implementation Details

The prototype essentially follows the same structure as presented in Chapter 9 where the reuse analysis phase is explicitly preceded by the liveness derivation phase w.r.t. the default liveness descriptions of the individual procedures. The prototype is also embedded in the AMAI-framework, and therefore, the analysis of one individual module follows the same scheme as for the non-modular analysis (Figure 9.2). The main difference between both systems is that here the analysis of a module also results in the generation of an interface file. These interface files are then used during the analysis of modules that depend on the modules from which they stem.

The analysis of one module consists of annotating the code with forward and (analysis-based) backward use information, and deriving the rule base meaning of its procedures in the context of a goal-independent structure sharing analysis. The results of these annotations and the structure sharing analysis are used for the derivation of the liveness information. Finally, the liveness results together with the structure sharing information, are used for detecting direct reuses and propagating them as indirect reuses. At the end of the analysis we obtain procedures that are fully annotated with reuse information. The generated interface file contains the local structure sharing information for each of the exported procedures, as well as the (compacted) reuse information tuples for the procedures that potentially allow some form of structure reuse. Figure 10.2 sketches the phases of the analysis starting from AMAI instructions that are generated from the original Mercury file, and ending with annotated code formatted as HTML-code. The arrows connecting each of the analysis steps represent the flow of information (e.g., liveness annotation requires forward use, backward use, as well as structure shar-

ing information). Dashed lines are used to represent the interaction with optimisation interface files, *i.e.*, the dependence of information stored in the interface files of other modules, and the optimisation interface file that is generated for the analysed module as a side-product of that analysis.

In this prototype, no actual versions of the procedures are generated as the results of the analysis are not fed back into the Mercury compiler.

10.5.4 Benchmarks and Results

The first type of benchmarks is a selection of library modules as they come with the Mercury distribution[6]. These modules are constantly used within any kind of application (including real-world applications) and the exploitation of the reuse opportunities will have a large impact in general. The library modules have many interdependencies and can not be analysed without a module based approach. We make a distinction between *basic* library modules, and the other library modules. Basic library modules are modules that only depend on themselves, and do not import any other modules.

A second kind of benchmarks are modules from real-world applications ranging from small stand-alone programs to some modules of the Melbourne Mercury compiler implementation.

The following modules were analysed:

- basic library modules for tree and list manipulation (`assoc_list`, `bintree`, `bool`, `bt_array`, `list`, `set_ordlist`, `tree234`);
- library modules that import procedures from the basic ones (`bag`, `bintree_set`, `eqvclass`, `graph`, `group`, `map`, `multi_map`, `queue`, `set`, `set_unordlist`);
- a module of the industrial users of Mercury in the ESPRIT project ARGo (`argo_cnters`) and
- and finally, some modules from the Mercury compiler (`labelopt`, `llds`, `opt_util`).

In Table 10.1 the library modules are in the upper part and the other modules in the lower part. Our focus goes mainly to exported procedures, as these are the only procedure visible outside of the modules, and for which analysis information is explicitly recorded in interface files.

The legend to Table 10.1 is as follows:

- **Time** The time in seconds of all the analyses performed on a module, including structure sharing, liveness analysis as well as the detection and propagation of reuse opportunities.

[6]The experiments date from January 2000 where we worked with the most recent release of the day of version 0.9.1 of the Melbourne Mercury Compiler.

- **Pr** The number of procedures in the module.
- **Xp** The number of exported procedures.
- **CR** The number of exported procedures for which conditional reuse is detected.
- **UR** The number of exported procedures for which unconditional reuse is detected. (The two types of reuse can occur in the same procedure.)
- **NR** The number of exported procedures without reuse.
- **Cnd** The average number of reuse information tuples for exported procedures with conditional reuse. Note that in this implementation the sets of reuse tuples may contain duplicates. The reduced sets are not available here.
- **DC** The number of matching deconstruction/construction pairs in the analysed module.
- **%DR** The percentage of matching deconstruction/construction pairs resulting in direct reuse.
- **Lc** The number of calls to procedures internal at the module.
- **%LR** The percentage of internal procedure calls that calls a version with reuse.
- **Ec** The number of calls to imported procedures.
- **EcR** The number of calls to imported procedures for which a reuse version exists.
- **%ER** The percentage of the calls to imported procedures for which a reuse version exists and which meets the reuse information tuples so that a call to the reuse version is safe. ("-" if EcR=0).

Our experiments were done on the same platform as used for our first prototype, *i.e.*, using an UltraSPARC-IIi (333Mhz) with 256MB RAM, using SunOS Release 5.7, under a usual (small) workload. The Prolog-engine used was Master Prolog, release 4.1 ERP.

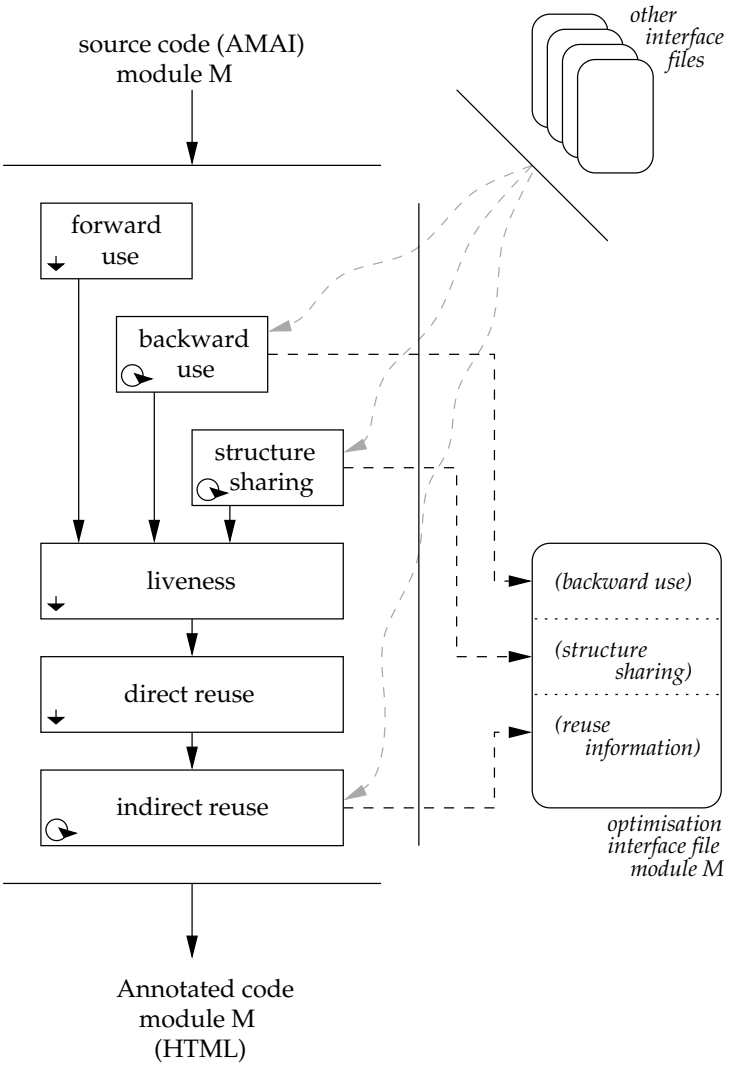


Figure 10.2: Sketch of the module-enabled prototype implementation. Parts requiring a fixpoint analysis are labelled with a loop, parts not requiring a fixpoint contain a straight downward arrow.

module	Time	Pr	Xp	CR	UR	NR	Cnd	DC	%DR	Lc	%LR	Ec	EcR	%ER
assoc_list	0.27	7	6	5	0	1	2.40	7	100	7	86	10	0	-
bag	2.46	26	24	17	10	5	4.95	4	100	29	90	70	25	100
bintree	1.37	30	19	6	2	11	3.00	17	100	56	43	33	0	-
bintree_set	0.16	23	21	11	6	9	1.08	1	100	16	75	26	6	100
bool	0.01	5	5	2	0	3	2.00	2	100	2	100	0	0	-
bt_array	5.97	37	17	6	4	9	1.75	38	97	137	55	181	0	-
eqvclass	1.55	21	12	4	4	4	1.00	5	100	27	78	37	12	100
graph	6.96	26	14	6	6	6	0.75	9	100	42	33	51	12	100
group	1.95	20	11	2	4	7	0.50	4	100	27	41	27	8	100
list	1.84	66	56	34	3	22	1.76	46	91	111	64	38	2	100
map	6.69	42	38	16	4	18	3.65	2	100	39	54	56	16	100
multi_map	1.95	36	33	11	6	16	2.82	1	0	13	46	48	21	100
queue	0.18	12	11	5	1	6	2.80	4	100	2	100	12	8	100
set	0.08	27	27	6	0	21	1.00	0	-	0	-	27	6	100
set_ordlist	0.52	31	29	6	0	23	1.00	7	0	41	2	19	6	100
set_unordlist	0.34	30	27	13	3	14	1.15	1	100	25	64	12	6	100
tree234	1140	74	20	10	1	9	5.55	283	88	443	71	136	0	-
argo_cnters	4.79	18	1	0	1	0	0.00	32	100	41	29	56	0	-
labelopt	12.00	6	2	2	1	0	2.00	2	100	17	88	13	9	78
llds	0.08	7	7	0	0	7	-	0	-	6	0	2	0	-
opt_util	2028	73	46	20	8	23	2.04	48	83	284	28	70	24	100

Table 10.1: Benchmark results; "-" means not applicable. See page 235 for the complete legend.

10.5.4.1 Discussion

- In most cases the analysis time is of the same order as the compilation time of the module. In a few cases it is rather large (`tree234` and `opt_util`). We do not consider these times as unbearable, especially for library modules as these often have to be compiled only once. This prototype leaves room for much improvement. Chapter 11 presents some improvements when presenting the practical aspects of integrating the reuse analysis into an existing Mercury compiler.

- The reuse analysis in this prototype is not fine-tuned enough to report reuse analysis times. In most cases the time needed for it is comparable to the time for the liveness analysis. We believe it should only be a small fraction of the total analysis time, especially in cases where the time needed for executing the liveness analysis is large.

- Reuse versions are created for a large fraction of exported predicates. Even unconditional reuse is quite frequent. This is an indication that our analysis is able to find an interesting amount of reuse.

- The average number of reuse information tuples is in general small. However for `bag` and `tree234` the average is about 5 reuse information tuples. In `tree234` there is a procedure with 12 reuse information tuples. Although some of these reuse tuples may appear to be equivalent, even upon simplification, it seems not feasible to create a specialised version for each of the combinations of reuse tuples. Chapter 13 proposes a system to evaluate the reuse opportunities of each procedure before actually producing the specialised versions for that procedure.

- Quite a large fraction of matching deconstruction/construction pairs result in direct reuse. If no direct reuse is detected, it is either because no reuse is possible or because the analysis is too imprecise. To find out what the real cause is of a *missed* reuse is a cumbersome task that needs to be done by hand and is therefore not feasible on a large scale.

- Versions of local procedures with reuse are quite frequently called.

- Although the total number of calls to imported procedures (Ec) looks high with regard to the number of calls to such procedures for which a reuse version exists (EcR), most of the calls happen to be I/O related, or integer-operations, hence are calls to procedures where reuse is not possible anyway.

10.5.4.2 Effect on the performance of Mercury programs

The results of this prototype are not fed back into the compiler, and therefore the effects on the performance of Mercury programs could at that stage only be guessed or had to be forced by manual intervention. Chapter 12 discusses the incorporation of the reuse analysis into the Mercury compiler. We discuss the performance effects there.

10.6 Conclusion

The contribution of this chapter is to develop a modular reuse analysis that can become part of a complete compile-time garbage collection system suited to compile and optimise individual modules w.r.t. the memory usage of their procedures. While this process was fairly straightforward for the liveness analysis — liveness analysis is fitted for goal independent based analysis—, it was a non-trivial task for reuse analysis. The major contributions are the introduction of so called *reuse information* and two theorems that allow to reason about this reuse information: how it is created, how it can be used to verify reuse, how it can be compacted, and how it can be propagated through procedure calls. As a result of the analysis of a module, the structure sharing and reuse information of the exported procedures are stored in an interface file. This information can be used to correctly analyse other modules that depend on it.

We adapted our prototype implementation of Chapter 9 and used it for a number of significant benchmarks: important library modules of the Melbourne Mercury compiler, a stand-alone program and some modules of the Melbourne Mercury compiler implementation. Although the results are not used to compile the modules into code that actually performs the reuses, we observed that the number of opportunities of reuse are significant, while the reuse information tuples restricting the use of these reuses are limited. This promises a fair potential of reuse.

The first steps of the theory in this chapter were presented in (Mazur, Janssens, and Bruynooghe 1999b) and (Mazur, Janssens, and Bruynooghe 1999c). This theory was perfected in (Mazur, Janssens, and Bruynooghe 2000) where also the results of the adapted prototype were discussed.

Chapter 11

Practical Aspects for a Working Compile-Time Garbage Collection System for Mercury

In previous chapters we have mainly dealt with the program analysis part of a compile-time garbage collecting (CTGC) system assuming that finding matching deconstruction/construction unifications is straightforward. But of course, the task of determining these assignments is an optimisation problem of its own.

Other practical problems that need to be tackled in order to obtain a working compile-time garbage collection system range from increasing the speed and precision of the underlying analyses to obtaining a better reuse behaviour by lifting the locality principle for data structure reuses.

We handle each of these practical aspects in the present chapter. The description of the implementation of the resulting CTGC system is postponed to the following chapter where also the benchmark results obtained with this implementation are discussed.

11.1 Reuse decisions

In the previous sections we did not go into the details of detecting matching deconstruction/construction pairs, but of course the strategy for detecting these pairs can have an influence on the reuse results of the CTGC system.

The following example illustrates how different matching deconstruction/-construction pairs can be identified, each yielding different memory reuse behaviour.

```
:- type field1 ---> field1(int, int, int).
:- type field2 ---> empty; field2(int, int).
:- type list(T) ---> [] ; [T | list(T)].
:- pred convert2(list(field1), list(field2)).
:- mode convert2(in, out) is semidet.

convert2(List0, List):-
    ( % switch on List0
        List0 => [], List  <= []
    ;
        (d1) List0 => [Field1 | Rest0],
        (d2) Field1 => field1(A, B, _C),
        (c1) Field2 <= field2(A, B),
        convert2(Rest0, Rest),
        (c2) List <= [Field2 | Rest]
    ).
```

Figure 11.1: Converting lists. An example of multiple choices for matching deconstruction/construction pairs. Only the program points of interest are explicitly annotated: (d_1), (d_2), (c_1) and (c_2). See Example 11.1.

Example 11.1 *Figure 11.1 shows the code for converting a list of items of type* field1 *into a list of items of type* field2*. The goal-independent (or default) liveness analysis of that procedure identifies the deconstructed data structures at d_1 and d_2 as available for reuse. The procedure also contains two constructions in which the memory from the dead cells could be reused. We can establish two different reuse schemes according to which data structures are reused by which new terms: the dead cells from d_1 are reused by construction c_1, and the cells from d_2 are reused in c_2, or the other way around. Or if reuse is restricted to data structures of the same arity, then only the deconstructed data structure from d_1 can be reused. In that case, we have the choice of reusing it for c_1 or c_2. Each of these combinations yields an acceptable reuse scheme, yet, which one is the most interesting? One can for example argue that reusing the list-cell for constructing the new list-cell can be more interesting than reusing it for a field-cell as such a reuse does not need to update the constructor-field. Another argument can favour the reuse of the* field1 *term for the* field2 *term, as here, none of the arguments of the functor need to be updated. In fact, such a reuse requires precisely one pointer change (letting variable Field2 point to the memory that was pointed at by variable Field1), and one functor-change. But this reuse has one drawback, namely that one of the arguments of* field1 *is left unused, possibly leading to a form of memory leakage.*

The general assignment problem can be characterised by the following elements (Debray 1993):

1. a set of *producers* (deconstructions) that produces the memory that can be reused;
2. a set of *consumers* (constructions) that consume memory;
3. a constraint that either allows a producer to be consumed by a set of consumers, or limits the reuse to a one-one mapping;
4. a gain that is associated with each producer-consumer (or set of consumers) pair and that reflects the gain of reusing the producer instead of constructing the data structure from *new* memory;

Using this information, the reuse decision problem can be seen as finding a mapping from producers to consumers (or sets of consumers) with maximal gain. The gain can either reflect the gain in execution time, or the gain in memory saving, or a combination of both, depending on the desired effect of the optimisation.

This general reuse problem is NP-complete, yet by restricting the reuse of a producer by at most one consumer the problem becomes polynomial. This restricted problem is called the *simple* reuse problem (Debray 1993) for which it has been shown that it can be reformulated as an instance of the maximum weight matching problem for a weighted bipartite graph. In the context of Mercury we slightly lift the constraint of only allowing one consumer for each producer in the sense that if multiple consumers are assigned to the same producer, then these consumers must be mutually exclusive, *i.e.*, they must be on different execution paths. A typical example is the occurrence of adequate consumers in each of the branches of a disjunction. When executed, at most one of these branches is followed, therefore, only one of the consumers will actually consume the memory freed by the producer. Note that the reuse problem remains simple, as it suffices to interpret a disjunction as one single consumer to obtain the initial simple reuse problem.

The *simple* reuse problem corresponds to our initial formulation of assigning the reuses where only matching deconstruction/construction pairs are considered.

11.1.1 Simplified Approach

In a first approach we address the problem of determining the producer-consumer pairs by simplifying the general matching problem to two orthogonal decisions: imposing constraints on the allowed reuses and using simple strategies to select amongst different possible candidates for reuse. We discuss each of these issues.

Constraints on allowed reuses. Constraints allow one to express common characteristics between the dead data structures and the new to be constructed data structures. They also reflect the restrictions that can be imposed by the back-end (c.f. Section 3.8.1) to which a Mercury program is compiled. For example, reusing a data structure corresponding to a term of arity 15 for a data structure of a term

of arity 2 might not be desirable if the garbage collector is not able to recover the 13 remaining memory-words. Or if changing the type of a data cell is impossible[1], then we can only allow reuses for matching constructors.

We have implemented the following constraints:

- *Almost matching arities.* This constraint expresses the intuition that it can be worthwhile to reuse a dead data structure, even if not all memory-words are reused. This is indeed interesting if it can be guaranteed that the superfluous words will be collected by the run-time garbage collector within a reasonable delay.

  In Example 11.1, allowing a difference of size one allows (c_1) and (c_2) to reuse the memory available from either (d_1) or (d_2).

- *Matching arities.* If the run-time system is not powerful enough to be used for the setting of *almost matching arities*, then a more restrictive constraint can be used: only allow reuse between data structures that have the same arity.

  This means that in our example only (d_1) can be reused (by either (c_1) or (c_2).

- *Label-preserving.* Using the Java or .NET back-end, it is not possible to change the type of run-time objects, therefore reuse is only allowed if the dead and new data structures have the same constructor. This type of restriction is commonly called *label-preserving* (Debray 1993; Gudjonsson and Winsborough 1993).

  In Example 11.1, we obtain that the data structure freed at (d_1) can only be reused in the construction at (c_2).

Selection strategies. When a dead data structure can be reused by different construction unifications, or when a construction unification has different dead structures at its disposition, a choice has to be made. In Example 11.1 (c_1) can either reuse the cell available from (d_1) or (d_2). Some choices yield better results than others. In order to keep it simple in this approach, we limit our selection criteria to one of the following:

- *Lifo.* Traverse the body of the procedure and assign the reuses using a last-in-first-out selection strategy. This means that when a choice is left for a given construction, choose the data structure that became available for reuse most recently. The intuition is that after deconstructing a variable, it is likely that a new similar structure will be constructed in the same context.

[1] This is the case when using .NET as the back-end for Mercury.

In Example 11.1, if (c_1) is allowed to reuse the data structure freed at (d_1) or (d_2), then according to this strategy, data structure $Field1^{\overline{\epsilon}}$ will be reused for constructing data structure $Field2^{\overline{\epsilon}}$, and $List0^{\overline{\epsilon}}$ for $List^{\overline{\epsilon}}$. For this particular example this corresponds to the best choice one can make w.r.t. the number of heap cells that need to be updated. Indeed, the reuse of $Field1^{\overline{\epsilon}}$ corresponds to a simple tag change on the pointer as a all the positions of the new data structure have the same value as the corresponding positions of the reused cell.

- *Random.* The intuition behind the lifo-strategy is not always true. Consider the following disjunction:

```
X => f(...),
( ... Y <= f(...)
; ... ),
Z <= f(...)
```

Here the data structure $X^{\overline{\epsilon}}$ can be reused either for constructing $Y^{\overline{\epsilon}}$ or for constructing $Z^{\overline{\epsilon}}$. As the first branch of the disjunction that the construction unification for Y is part of is not guaranteed to be executed for each program call, it is more interesting to allow $X^{\overline{\epsilon}}$ to be reused for constructing $Z^{\overline{\epsilon}}$ instead of constructing $Y^{\overline{\epsilon}}$.

Therefore we add a simple selection strategy that randomly selects the dead data structures amongst all the available candidates.

11.1.2 Constructing Graphs

Although the results obtained with the simplified orthogonal choices presented above are already interesting, we also provide a graph based approach. In this approach, the possibilities for reuse are presented as a weighted graph as suggested by Debray (1993), while the search for an optimal solution is replaced by a heuristic approach.

The goal is to assign consumers to producers in a so called reuse mapping. For this purpose we generate a table where each entry consists of a data structure available for reuse, a value and a list of constructions that can possibly reuse that data structure. The idea is that the value reflects the gain that the reuse may bring. The data structure with highest value is selected and removed from the table. Its corresponding information is recorded in the final reuse mapping, and the involved constructions are annotated with the information that they will be reusing the data structure in question. This process is repeated until no reusable data structures are left.

The value of a data structure that becomes available for reuse at a given deconstruction unification is computed by taking into account the call graph of the

procedure, simplified such that only construction unifications, conjunctions and disjunctions are reflected. The root of the graph is a deconstruction unification, the leaves are construction unifications. The branches are either conjunctions or disjunctions. The value of each of the nodes of the graph, including the root node, is computed using the following rules:

- If the node is the root of a conjunction, then the value of that node is the maximum of the values of the nodes depicting the branches of the conjunction. Intuitively, the idea is that if a data structure is followed by a conjunction of two constructions that can potentially reuse the dead heap cells, then it is better to choose the construction with highest value.

- If the node is at the source of a disjunction, then the value of that node is taken to be the average of the values of the nodes of the branches of that disjunction. This reflects the intuition that if the deconstruction of a data structure is followed by a disjunction with two branches in which reuse is only allowed within one of its branches, then the global effect is that the structure reuse can only be realised in a fraction of the calls to this disjunction. Assuming that each branch within a disjunction has the same probability of being selected, the gain of allowing reuse in one of the branches must be computed as the average of the gains of the branches in that disjunct.

- If the node is a construction unification then the value reflects the gain of reusing the dead data structure at the root of the graph w.r.t. allocating fresh memory for that construction. We assume that only construction unifications are considered that *can* potentially reuse the dead data structure[2]. If the functors differ, then we need to take into account the extra cost of updating the functor. If the arities differ, then we add a penalty cost that reflects the gain one would have had if the unusable heap cells would have been reused too. If some of the fields of the dead data structure do not need an update, then this is added as extra gain.

  Hence, to evaluate the gain, which we denote by v, we need the following information:

  - if the constructed structure and the dead structure have the same constructor then the parameter c takes the boolean value true, else it takes the value false.
  - the arity of the constructed structure, a_c.
  - the arity of the deconstructed structure, a_d. Note that a new structure can only reuse an old structure if $a_c \leq a_d$.

[2]This means that if the back-end does not allow to reuse data structures with differing functors, then constructions violating that rule are immediately discarded from the graph.

– the number of fields that needs to be updated if the construction reuses the available dead structure, u.

Then the gain can be computed using the following formula:

$$\begin{array}{rcl} v & = & \text{cost for creating new cell} \; - \; \text{cost for updating dead cell} \\ & = & ((\alpha+\gamma)\cdot a_c+\beta)-(\gamma\cdot u+(c?0;\beta)+\alpha\cdot(a_d-a_c)) \end{array} \tag{11.1}$$

where α represents the cost of allocating fresh heap cells, γ is the cost of updating heap cells, and β is the cost of updating the constructor information of the data structure. We use the notation $c?0;\beta$ to express the conditional expression "if c is true, then use the value 0, or else take β". The last term, *i.e.*, $\alpha\cdot(a_d-a_c)$, in the equation reflects a penalty for not reusing each of the heap cells of the dead structure assuming that the cost of producing the garbage cells is the same as the cost of creating new heap cells.

The actual values of these cost parameters are determined *ad hoc*. In the implementation we used the values

$$\begin{array}{rcl} \alpha & = & 5 \\ \beta & = & 1 \\ \gamma & = & 1 \end{array}$$

reflecting the number of instructions to allocate new memory, initialise the functor information and update the heap cells respectively.

We illustrate the process with some examples.

Example 11.2 *Using the type-declarations of Example 11.1, we write the following procedure:*

```
%:- pred transform(field1,field1,field2).
%:- mode transform(in,out,out) is det.
transform(Input, Out1, Out2):-
    (d1) Input => field1(A,B,C),
    (
        A == 0
    ->
        (c1) Out2 <= field2(B,C)
    ;
        (c2) Out2 <= empty
    ),
    (c3) Out1 <= field1(C,B,A).
```

Intuitively, it is more interesting to reuse the deconstructed field1 *term for the new* field1 *term, instead of reusing it for either of the two other constructions. Using the lifo approach and allowing differing arities, the reuse mapping would assign* (c_1) *for reusing*

the dead cells of (d_1)*. Using the graph based approach, the value of the deconstruction is* 17*, and the most interesting reuse opportunity is determined to be* (c_3)*. The graph that is conceptually built is shown in Figure 11.2. Note that the if-then-else construct is represented as a disjunction of the conjunction of the test and the then-branch, and the else-branch.*

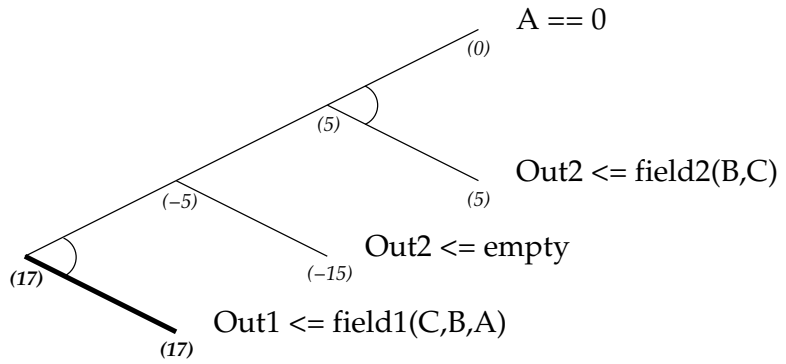


Figure 11.2: Gain Graph for the reuse of the structure field1(A, B, C) in Example 11.2. Conjunctions are differentiated from disjunctions by the use of extra arcs joining the branches. The values of the nodes are computed using Equation (11.1) with $\alpha = 5, \beta = 1$ and $\gamma = 1$. The branch yielding the maximal gain is shown in bold.

Example 11.3 *Using the type declaration*

```
:- type t ---> f(int, int); g(int, int).
```

we write the following procedure:

```
%:- pred transform(t, t, t).
%:- mode transform(in, in, out) is semidet.
transform(T0, T1, T):-
        T0 => f(A, _B),
        T1 => g(C, _D)
        T <= f(A, C).      % reuse T0
```

The gain graphs for $T0$ *and* $T1$ *are graphs that contain only one edge, namely to the construction unification* `T <= f(A,C)`*. For* $T0$ *the gain is valued to be* $12 = (6 \cdot 2) + 1 - (1 \cdot 1 + 0 + 5 \cdot 0)$*, while the gain for* $T1$ *is* $10 = (6 \cdot 2) + 1 - (1 \cdot 2 + 1 + 5 \cdot 0)$*. Hence, the deconstructed data structure* $T0^{\overline{e}}$ *is more interesting for the construction of* T *then the data structure* $T1^{\overline{e}}$*.*

When two dead data structures turn up having the same assigned value, then this means that both data structures yield the same supposed gain, hence the algorithm could choose any of these structures.

Example 11.4 *Using the type declaration of Example 11.3, we define the procedure:*

```
%:- pred transform(t,t,t,t).
%:- mode transform(in,in,out,out) is semidet.
transform(In1, In2, Out1, Out2):-
        In1 => f(A,B),
        Out1 <= f(B,A),
        In2 => f(C,D),
        Out2 <= f(D,C).
```

*Taking into account that the code is not reordered, and that the left-to-right execution scheme is held, Out*1 *can reuse In*1*, and Out*2 *can either reuse In*1 *or In*2*. For both data structures our algorithm will derive the same value. If we make a bad choice, and let In*1 *be reused by Out*2 *then no reuse is possible for Out*1*.*

In the above example, clearly, *Out*2 should reuse *In*2 while *Out*1 should reuse *In*1. To deal with such cases in our algorithm we add a notion of *degree*. The degree of a deconstructed cell is the number of separate constructions by which it could be reused. We then change our algorithm in such a way that instead of picking the data structure with maximal value v, we choose the data structure with maximal value-degree ratio, *i.e.*, for which the result v/d is maximal, where d is the degree of the structure. The intuition is that if two data structures have the same value, then the one with the smallest number of opportunities to be reused should be assigned first. In the previous example, the degree for *In*1 is two, while the degree for *In*2 is only one. Hence, *In*2 will be selected first, and is assigned to being reused by *Out*2. In a second iteration, *In*1 can be assigned to being reused by *Out*1.

Note that establishing graphs for assigning construction unifications to dead data structures is a selection strategy just like the *lifo* and *random* criteria are selection strategies (Section 11.1.1). This means that constraints on the allowed reuses can also be relevant in this context. For example, if the arities of the involved data structures must be the same, then the constructions of data structures with different arities are simply not even considered in the graph, similarly for other constraints.

11.1.3 Related Work

Most of the research in the area of compile-time garbage collection has focused on the liveness analysis aspects involved with it. To the best of our knowledge,

basically only two authors have recognised the problem of finding the best assignments for the dead data structures, hence tackle the actual optimisation of the generated code.

In (Debray 1993) the authors define the reuse optimisation problem, prove that it is NP-complete and produce a nearly optimal strategy for solving the general reuse problem. They also tackle the simple reuse problem where a producer can only be reused by at most one consumer, and prove that it is an instantiation of the maximum weight matching problem for a weighted bipartite graph. The weights can be adjusted for minimising memory usage or execution time. We used this graphing approach in our implementation, yet without actually solving the maximum weight matching problem, being satisfied with the nearly optimal solution rendered by our algorithm.

In (Gudjonsson and Winsborough 1993) the focus of the memory reuse aspect is mainly on the execution time to be minimised. The idea is to try to discover almost every heap cell not requiring an update, going even beyond the boundaries of single procedures. This may indeed be important in Prolog, where the determinism of procedures is not necessarily known at analysis time, and where given the underlying data-flow analysis, each cell update requires extra care in the case the value has to be reset upon backtracking. In Mercury, where determinism *is* known at compile-time, and where the analysis explicitly takes into account backtracking, this is not a major issue. Therefore, it is not our immediate intention to try to avoid every possible cell update.

11.2 Enhancing the Structure Sharing Precision

In general, during the analysis of a typical program, it may be possible to encounter procedure calls for which the analysis will not be able to derive the structure sharing they build up. This is the case for procedures defined in terms of foreign code (c, C++, Java), higher-order calls and type-classes, but it can also be due to calls to imported procedures from modules that have not yet been analysed and for which no interface files have been generated yet (mutual dependent modules).

In the absence of any further information, we must approximate this unknown structure sharing by the top element from the abstract structure sharing domain, *i.e.*, the presence of all possible structure sharing between the variables involved. We abbreviate this abstract element to top. This is a perfectly correct and safe approximation as it expresses the total lack of knowledge about the possible existing structure sharing at some program point in the program. Unfortunately, once the analysis encounters a top approximation, all subsequent descriptions will also result in top as all operations combining structure sharing sets produce top once one of the arguments is top.

To obtain a usable CTGC system we need techniques to limit the creation and

propagation of top in the structure sharing analysis parts. In our implementation we use the following three techniques: using heuristics, adding a pragma to be able to manually annotate the critical code, and iterate the compilation for mutually dependent modules. We discuss each of these techniques.

Using heuristics. Based on the type- and mode- declaration of a procedure, one can derive whether it can create shared data structures or not, without looking at the procedure's body. This is the case when a procedure uses unique objects (declared `di` or `uo`), or only has unique output variables, or when the non-unique output arguments are of a type for which sharing is not possible (integers, enums, chars, etc.). In all these cases, it is safe to conclude that the procedure does not create additional structure sharing. Note that a procedure call can create new structure sharing between input variables as they must be ground at the moment when the procedure is called.

Manual structure sharing annotation for foreign code. Important parts of the Mercury Standard Library consist of procedures that are defined in terms of foreign code. With the intention to be used mainly in this standard library, we have extended the Mercury language such that foreign code can be manually annotated with structure sharing-information.

Manual iteration for mutual dependent modules. At the time of implementing the CTGC system into the Melbourne Mercury compiler, this compiler was not yet able to cope with mutual dependent modules[3]. Consider a module A in which some procedures are expressed in terms of procedures declared in a module B, and vice versa. The normal compilation scheme is to compile one of the files, and then the other one. In the presence of an optimising compiler this is not enough. At the moment the first module is compiled, nothing is known from the second one, yielding bad precision for the first one. This bad precision propagates further to the second file as the second file relies on the first one.

As a work around we provide a way to manually control the incremental compilation of a program in which mutually dependent modules occur.

11.3 Making Analysis Faster: Widening the Structure Sharing

While it is interesting to have more precise structure sharing information than simply top, having more aliases also slows down the system. Now one can argue

[3] In the meantime this has changed, and a more advanced compilation scheme has been implemented into the MMC, roughly following the ideas presented in (Bueno, García de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2001). Our CTGC system has not yet been adapted accordingly.

that speed is not a major requirement of a CTGC system as it is primarily intended to be used only at the final compilation phase of a program, but even for our benchmarks we were not ready to wait hours for a module to compile. Therefore, in order to produce a usable CTGC system we add a widening operator (Cousot and Cousot 1992c) that acts upon the aliases produced. This widening operator can be enabled on a per-module base. The user can also specify the threshold at which widening should be performed: e.g. only widen if the size of the structure sharing set exceeds 1000.

Recall that during structure sharing analysis, a data structure is represented by its full path down the term it is part of. Such a path is a concatenation of selectors that selects the functor and the exact argument position in the functor. Structure sharing is then expressed as a pair of data structures.

For the structure sharing, we introduce a so called *type widening*. The key idea is to replace a path of individual selectors by one single selector that represents the type of the type node that is selected by that path of selectors[4]. The following example illustrates the intuition behind this form of widening.

Example 11.5 *Consider the type definition*

```
:- type tree(T) --->
      empty;
      two(T, tree, tree);
      three(T, T, tree, tree, tree).
```

and the conjunction of construction unifications:

```
L <= [1,2,3],
A <= two(L,empty,empty),
B <= two(L,A,A),
C <= three(L,L,A,B,A)
```

At the end of these four unifications, we obtain the structure sharing set as depicted in Figure 11.3. While all the structure sharing relating subtrees of A, B and C are simply reduced to structure sharing pairs between $A^{\overline{\epsilon}}$, $B^{\overline{\epsilon}}$ and $C^{\overline{\epsilon}}$, on the other hand, all the structure sharing involving the sharing of the common element L are kept explicit[5]*. This has the end effect that 28 of the 33 sharing pairs are related to that common element.*

Now imagine that C, the variable having the most internal structure sharing pairs, would be involved with subsequent unifications, then the number of structure sharing relations can become really big. The widening we consider is to replace each sequence of

[4]Applying type widening at each step of the analysis ultimately leads to a so called *type based* analysis approach, c.f. Section 12.7.

[5]Note that the sharing pairs 28-33 may seem unusual as C is not bound to a term with outermost functor *two*/3. Recall that in the abstract representation of data structures we use the equivalence classes of selectors instead of selectors. This means that for elements of type tree(T), the selectors $(two, 2)$, $(two, 3)$, $(three, 3)$, $(three, 4)$, $(three, 5)$ are all equivalent to the empty selector ϵ, therefore, a selector such as $(three, 3) \cdot (two, 1)$ can be simplified to $(two, 1)$.

1. $\left(A^{\overline{(two,1)}} - L^{\overline{\epsilon}}\right)$
2. $\left(B^{\overline{\epsilon}} - A^{\overline{\epsilon}}\right)$
3. $\left(B^{\overline{\epsilon}} - B^{\overline{\epsilon}}\right)$
4. $\left(B^{\overline{(two,1)}} - L^{\overline{\epsilon}}\right)$
5. $\left(B^{\overline{(two,1)}} - A^{\overline{(two,1)}}\right)$
6. $\left(B^{\overline{(two,1)}} - B^{\overline{(two,1)}}\right)$
7. $\left(C^{\overline{\epsilon}} - A^{\overline{\epsilon}}\right)$
8. $\left(C^{\overline{\epsilon}} - B^{\overline{\epsilon}}\right)$
9. $\left(C^{\overline{\epsilon}} - C^{\overline{\epsilon}}\right)$
10. $\left(C^{\overline{(three,1)}} - L^{\overline{\epsilon}}\right)$
11. $\left(C^{\overline{(three,1)}} - A^{\overline{(two,1)}}\right)$
12. $\left(C^{\overline{(three,1)}} - B^{\overline{(two,1)}}\right)$
13. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,2)}}\right)$
14. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,3)\cdot(two,1)}}\right)$
15. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,5)\cdot(two,1)}}\right)$
16. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,4)\cdot(two,1)}}\right)$
17. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,4)\cdot(two,2)\cdot(two,1)}}\right)$
18. $\left(C^{\overline{(three,1)}} - C^{\overline{(three,4)\cdot(two,3)\cdot(two,1)}}\right)$
19. $\left(C^{\overline{(three,2)}} - L^{\overline{\epsilon}}\right)$
20. $\left(C^{\overline{(three,2)}} - A^{\overline{(two,1)}}\right)$
21. $\left(C^{\overline{(three,2)}} - B^{\overline{(two,1)}}\right)$
22. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,2)}}\right)$
23. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,3)\cdot(two,1)}}\right)$
24. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,5)\cdot(two,1)}}\right)$
25. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,4)\cdot(two,1)}}\right)$
26. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,4)\cdot(two,2)\cdot(two,1)}}\right)$
27. $\left(C^{\overline{(three,2)}} - C^{\overline{(three,4)\cdot(two,3)\cdot(two,1)}}\right)$
28. $\left(C^{\overline{(two,1)}} - L^{\overline{\epsilon}}\right)$
29. $\left(C^{\overline{(two,1)}} - A^{\overline{(two,1)}}\right)$
30. $\left(C^{\overline{(two,1)}} - B^{\overline{(two,1)}}\right)$
31. $\left(C^{\overline{(two,1)}} - C^{\overline{(three,1)}}\right)$
32. $\left(C^{\overline{(two,1)}} - C^{\overline{(three,2)}}\right)$
33. $\left(C^{\overline{(two,1)}} - C^{\overline{(two,1)}}\right)$

Figure 11.3: Complete list of the abstract structure sharing pairs obtained for the code of Example 11.5.

selectors by a selector reflecting the type of the involved type nodes. This would mean that all the selectors related to the sharing of L are then reduced to one single selector, namely a selector representing the type of L, i.e., *in this case the polymorphic type variable* T.

11.3.1 T-selectors

We extend the set of selectors in *Selector* to include types. Let *TSelector* denote the set of all sequences over $(\Sigma \times \mathbb{N}) \cup \mathcal{T}(\Sigma_{\mathcal{T}}, \mathcal{V}_{\mathcal{T}})$. Elements from *TSelector* are called *t-selectors*. To make a distinction between normal selectors, type selectors are written in bold face and superscripted with the symbol $\sharp$, e.g. $\boldsymbol{s}^{\sharp}, \boldsymbol{s_X}^{\sharp}, \ldots$. T-selectors can be concatenated as usual ($\boldsymbol{s_1}^{\sharp}, \boldsymbol{s_2}^{\sharp} \in \mathit{TSelector} : \boldsymbol{s_1}^{\sharp} \bullet \boldsymbol{s_2}^{\sharp}$), and $\boldsymbol{\epsilon}^{\sharp}$ is the empty t-selector and neutral element for the concatenation. The intuition behind the use of t-selectors is that instead of selecting a type node using an exact path to that type node, t-selectors select their type nodes by the types of these type nodes.

Example 11.6 *Consider the type* $\mathsf{list}(\mathsf{T})$ *defined in the usual way, then the t-selector* $\mathbf{T}^{\sharp}$ *selects all the type nodes of type* T *from the type tree of* $\mathsf{list}(\mathsf{T})$. *The t-selector* $(\mathbf{[|]}, \mathbf{2})^{\sharp} \cdot \mathbf{T}^{\sharp}$

selects all the type nodes of type T from the first sublist of the type $\mathsf{list}(\mathsf{T})$.

In fact, each t-selector in *TSelector* represents a set of selectors in *Selector*. This relation is formalised by defining a mapping relation between *TSelector* and $\wp(\textit{Selector})$. We first define the notion of a normal selector being *covered* by a t-selector.

Definition 11.1 (Covering) *Let* $\boldsymbol{s}^\sharp = \boldsymbol{s_1}^\sharp \cdot \boldsymbol{s_2}^\sharp \cdot \ldots \cdot \boldsymbol{s_n}^\sharp \in \textit{TSelector}$, *and* $s \in \textit{Selector}$ *be a t-selector and selector resp. for a type* t, *then* s *is* covered *by* $\boldsymbol{s}^\sharp$ *in the context of that type* t, *denoted by* $s \lhd_t \boldsymbol{s}^\sharp$ *iff* s *is a valid selector for* t *and one of the following situations holds:*

- *if* $\boldsymbol{s}^\sharp = \boldsymbol{\epsilon}^\sharp$ *then* $s = \epsilon$;
- *if* $\boldsymbol{s_1}^\sharp \in \mathcal{T}(\Sigma_\mathcal{T}, \mathcal{V}_\mathcal{T})$, *then* $s \lhd_\mathsf{t} \boldsymbol{s}^\sharp$ *iff* $\exists s_a, s_b \in \textit{Selector}$ *such that* $s = s_a \bullet s_b$, $\mathsf{t}^{s_a} = \boldsymbol{s_1}^\sharp$ *and* $s_b \lhd_{\boldsymbol{s_1}^\sharp} \boldsymbol{s_2}^\sharp \cdot \ldots \cdot \boldsymbol{s_n}^\sharp$;
- *if* $\boldsymbol{s_1}^\sharp \in \Sigma \times \mathbb{N}$, *then* $s \lhd_\mathsf{t} \boldsymbol{s}^\sharp$ *iff* $\exists s' \in \textit{Selector}$ *where* $s = \boldsymbol{s_1}^\sharp \bullet s'$ *and* $s' \lhd_{\mathsf{t}^{s_1}} \boldsymbol{s_2}^\sharp \cdot \ldots \boldsymbol{s_n}^\sharp$.

When the context of the type is clear, we abbreviate $s \lhd_\mathsf{t} \boldsymbol{s}^\sharp$ to $s \lhd \boldsymbol{s}^\sharp$.

Example 11.7 *In the context of the type* $\mathsf{list}(\mathsf{T})$ *we have for example*

$$
\begin{array}{l}
([|], 1) \lhd \mathbf{T}^\sharp \\
([|], 2) \cdot ([|], 1) \lhd \mathbf{T}^\sharp \\
\epsilon \lhd \mathbf{list}(\mathbf{T})^\sharp \\
\epsilon \lhd \boldsymbol{\epsilon}^\sharp \\
([|], 2) \cdot ([|], 2) \lhd \mathbf{list}(\mathbf{T})^\sharp
\end{array}
$$

but also

$$
\begin{array}{l}
([|], 2) \cdot ([|], 2) \lhd \mathbf{([|], 2)}^\sharp \cdot \mathbf{list}(\mathbf{T})^\sharp \\
([|], 2) \lhd \mathbf{([|], 2)}^\sharp \cdot \mathbf{list}(\mathbf{T})^\sharp
\end{array}
$$

The latter can be verified by observing that $([|], 2)$ *could be rewritten as* $([|], 2) \cdot \epsilon \cdot \epsilon$, *whereas* $\mathbf{([|], 2)}^\sharp \cdot \mathbf{list}(\mathbf{T})^\sharp$ *may be seen as* $\mathbf{([|], 2)}^\sharp \cdot \mathbf{list}(\mathbf{T})^\sharp \cdot \boldsymbol{\epsilon}^\sharp$.

Definition 11.2 (Mapping of a t-selector) *Consider* $\boldsymbol{s}^\sharp = \boldsymbol{s_1}^\sharp \cdot \boldsymbol{s_2}^\sharp \cdot \ldots \cdot \boldsymbol{s_n}^\sharp \in \textit{TSelector}$ *and* $S \in \wp(\textit{Selector})$ *in the context of a type* t, *then* $\boldsymbol{s}^\sharp$ *is* mapped *to the set of selectors* S *in the context of* t, *denoted by* $\boldsymbol{s}^\sharp \leftrightsquigarrow_\mathsf{t} S$, *iff* $\forall s \in S : s \lhd_\mathsf{t} \boldsymbol{s}^\sharp$ *and* $\forall s \in \textit{Selector} : s \lhd_\mathsf{t} \boldsymbol{s}^\sharp \Rightarrow s \in S$. *In such cases we also say that* S *is a mapping of* $\boldsymbol{s}^\sharp$ *for type* t.

Again, if the context is clear, then the subscript designating the type is omitted.

In the case of recursive types the mapping of a t-selector can be an infinite set. This is illustrated by the following example:

Example 11.8 *The mapping of the t-selector* $\mathbf{list(T)}^\sharp$ *in the context of the type* $\mathsf{list(T)}$ *is the (infinite) set of selectors that select a type node of type* $\mathsf{list(T)}$*:*

$$\{\epsilon, ([|], 2), ([|], 2) \cdot ([|], 2), ([|], 2) \cdot ([|], 2) \cdot ([|], 2), \ldots\}$$

We introduce the notion of a valid t-selector for a type:

Definition 11.3 (Valid t-selector) *A t-selector* $s^\sharp$ *is a* valid *t-selector for a type* t *iff its mapping S is not the empty set. Thus:* $s^\sharp \leftrightsquigarrow_t S \Rightarrow S \neq \{\}$.

Indeed, the mapping of a t-selector can only contain valid selectors in the context of a specific type, hence, if the mapping is empty, then the t-selector is considered to be invalid for that type.

Corollary 11.1 *If S is a mapping of a t-selector* $s^\sharp$ *for a type* t*, then S is a subset of the type tree of that type:* $s^\sharp \leftrightsquigarrow_t S \Rightarrow S \subseteq \mathcal{TT}_t$.

Of course, t-selectors can also be applied to terms in which case a set of subterms is selected instead of one single subterm. We can again use the notion of a *valid t-selector* for a term τ in the sense that each selector in the mapping of that t-selector must be a valid selector for τ.

11.3.2 Equivalence classes for t-selectors

Just as the number of valid selectors for a given type can be infinite for recursive types, also the number of valid t-selectors can be infinite. For the latter selectors to be usable in the context of an abstract analysis domain, we need a finite representation. We do this in the same way as we did for normal selectors (page 111), namely by introducing an equivalence relation between t-selectors, dividing the space of valid t-selectors of a type into equivalence classes according to the equivalence relation, and using one representative t-selector for each of the equivalence classes instead. Again, using the restriction that Mercury types are types for which the type nodes of these types are a finite set, we can guarantee that this equivalence relation partitions the t-selectors in a finite number of equivalence classes.

Definition 11.4 (T-selector equivalence) *The equivalence relation for t-selectors is similar to the equivalence of normal selectors. Formally, let* $s_1{}^\sharp, s_2{}^\sharp \in$ *TSelector applicable to a type* t*, then* $s_1{}^\sharp$ *is equivalent to* $s_2{}^\sharp$*, denoted by* $s_1{}^\sharp \equiv s_2{}^\sharp$*, iff,* $t^{s_1{}^\sharp} = t^{s_2{}^\sharp}$ *and* $\exists s^\sharp \in$ *TSelector* : $s_1{}^\sharp \bullet s^\sharp = s_2{}^\sharp$ *or* $s_2{}^\sharp \bullet s^\sharp = s_1{}^\sharp$

We use the same notation to denote the equivalence class or minimal element of that class of a t-selector as the notation we introduced for normal selectors. Let

$s^\sharp$ be a valid t-selector for a type t, then the equivalence class is written as $[s^\sharp]_\mathsf{t}$ and the minimal element of that equivalence class is written as $\overline{s^\sharp}_\mathsf{t}$. Formally:

$$\begin{array}{lcl} [s^\sharp]_\mathsf{t} & = & \{s'^\sharp \mid s'^\sharp \equiv s^\sharp\} \\ \overline{s^\sharp}_\mathsf{t} & = & s'^\sharp \in [s^\sharp]_\mathsf{t} \text{ such that } \forall s''^\sharp \in [s^\sharp]_\mathsf{t} : s''^\sharp = s'^\sharp \bullet e^\sharp, \text{ for some } e^\sharp \in \mathit{TSelector}. \end{array}$$

If the type context is clear, we drop the explicit subscript in the above notation.

Example 11.9 *For the type* $\mathsf{list}(\mathsf{T})$ *the t-selectors* $\epsilon^\sharp$, $\mathbf{list}(\mathbf{T})^\sharp$, $\mathbf{list}(\mathbf{T})^\sharp \cdot \mathbf{list}(\mathbf{T})^\sharp$ *are all equivalent. The minimal element of the equivalence class to which these types belong is* $\epsilon^\sharp$.

We define the notions of *covering* and *mapping*:

Definition 11.5 *Let* $s^\sharp \in \mathit{TSelector}$ *and* $s \in \mathit{Selector}$*, then* s *is* covered *by* $\overline{s^\sharp}$*, denoted with the same symbol* $s \lhd \overline{s^\sharp}$*, iff* $\exists s'^\sharp \in [s^\sharp]$ *such that* $s \lhd s'^\sharp$*. The* mapping *of a minimal element of a t-selector is a set* $S \in \wp(\mathit{Selector})$*, denoted by* $\overline{s^\sharp} \leftrightsquigarrow S$*, where* $S = \bigcup\{S' \mid s'^\sharp \in [s^\sharp], s'^\sharp \leftrightsquigarrow S'\}$, i.e., *S is the union of all the mappings of the selectors belonging to the same type-class as* $s^\sharp$.

Applying a minimal element $\overline{s^\sharp}$ to a type t (or term τ) now selects all the type-nodes (respectively subterms) designated by selectors covered by any of the elements from the equivalence class $[s^\sharp]$. T-selectors, or their minimal elements, can also be applied to variables in which case their meaning consists of the set of subterms of the terms to which these variables are pointing.

11.3.3 Data Structures, Sharing Sets and their Operations

We redefine data structures and structure sharing sets in the sense that every occurrence of a normal selector or an equivalence class over *Selector* is now replaced by a t-selector, resp. equivalence class over *TSelector*. The key difference is that the set of heap cells that these data structures (and structure sharing pairs) designate will be larger.

Example 11.10 *Consider a variable X with the type* $\mathsf{tree}(\mathsf{T})$ *defined in Example 11.5, then* $X^{\overline{(two,1)}}$ *specifically selects the first argument of the terms with outermost functor two bound to that variable, while* $X^{\overline{\mathbf{T}^\sharp}}$ *selects all the subterms with type* T *of the terms bound to X.*

This change implies some modifications on the operations defined on these entities, especially the definition of termshift. The purpose of the termshift operation in the context of concrete and abstract data structures is to obtain the full set

of data structures of all the subterms of these concrete or abstract data structures. Similarly, for concrete or abstract sharing pairs, the goal of termshift is to make all the structure sharing that they imply on the subterms of these terms explicit. This enabled us to order data structures and structure sharing pairs written in terms of the normal selectors simply by the set-inclusion operation w.r.t. the termshift operation. Thus, when data structures are expressed in terms of selectors from *Selector*, then all the data structures that they designate can be selected by concatenating the appropriate selector to the concerned selectors.

For data structures expressed in terms of t-selectors, the termshift operation becomes more complicated: to obtain all the data structures that are designated by one single data structure written in terms of a t-selector, we need to compute the mapping of the involved t-selector, and then compute the normal termshift on the result. The result of a termshift of data structures written in terms of t-selectors is a set of data structures over the usual domain of selectors, namely *Selector*. To distinguish the termshift over *Selector* from the termshift-operation over *TSelector*, we write the latter as $\mathbf{termshift}^\sharp$.

Definition 11.6 (Termshift for Concrete Data Structures) *Consider the set of concrete data structures $\langle e, D\rangle \in \langle Eqn^+, \wp(\mathcal{D}_{VI})\rangle$, then:*

$$\begin{aligned}&\mathbf{termshift}^\sharp(\langle e, D\rangle)\\ &= \bigcup\left\{\mathsf{termshift}(\langle e, \{X^s\}\rangle) \;\middle|\; X^{s\boldsymbol{x}^\sharp} \in D, \boldsymbol{s_X}^\sharp \leftrightsquigarrow_{\mathtt{type}(X^{s\boldsymbol{x}^\sharp})} S_X, s \in S_X\right\}\end{aligned}$$

For abstract data structures we have:

Definition 11.7 (Termshift for Abstract Data Structures) *Let $AD \in \wp(\overline{\mathcal{D}}_{VI})$, then*

$$\mathbf{termshift}^\sharp(AD) = \bigcup\{\mathsf{termshift}(X^{\bar{s}}) \mid X^{\overline{s\boldsymbol{x}^\sharp}} \in AD, \boldsymbol{s_X}^\sharp \leftrightsquigarrow_{\mathtt{type}(X^{\overline{s\boldsymbol{x}^\sharp}})} S_X, s \in S_X\}$$

In fact, each of the above definitions computes the mapping for each of the involved t-selectors, and performs the usual termshift on the selected data structures.

The definitions for termshifting concrete/abstract structure sharing pairs is similar to the above definitions.

The ordering of data structures and structure sharing pairs must also be revised. For simple data structures and structure sharing pairs it sufficed to compare one single data structure or structure sharing pair with a set of data structures or sharing pairs using the set-inclusion operation w.r.t. the termshifted set of data structures or sharing pairs. As a data structure or structure sharing pair defined in terms of t-selectors represents a set of such entities, a data structure or structure sharing pair is subsumed if and only if the termshift of that data structure or structure sharing pair is a subset of the termshift of the set of entities with

$$
\begin{array}{lll}
1. \left(A^{\overline{\mathbf{T}}^\sharp} - L^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 1 \\
2. \left(B^{\overline{\epsilon}^\sharp} - A^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 2, 5 \\
3. \left(B^{\overline{\epsilon}^\sharp} - B^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 3, 6 \\
4. \left(B^{\overline{\mathbf{T}}^\sharp} - L^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 4 \\
5. \left(C^{\overline{\epsilon}^\sharp} - A^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 7, 11, 20, 29 \\
6. \left(C^{\overline{\epsilon}^\sharp} - B^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 8, 12, 21, 30 \\
7. \left(C^{\overline{\epsilon}^\sharp} - C^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 9, 13 - 18, 22 - 27, 31 - 33 \\
8. \left(C^{\overline{\mathbf{T}}^\sharp} - L^{\overline{\epsilon}^\sharp}\right) & \leftarrow & 10, 19, 28
\end{array}
$$

Figure 11.4: Abstract structure sharing pairs from Example 11.5 after type widening (c.f. Example 11.11). The last column lists the original sharing pairs of Figure 11.3 from which each of the entries stems.

which it is compared. We use the same notation for ordering data structures, resp. structure sharing pairs expressed in terms of t-selectors as data structures, resp. structure sharing pairs expressed in terms of normal selectors.

Finally, as the definition of the ordering is slightly altered, the explicit variants of extend (Lemma 8.1) and extend_a (Definition 8.11) do not hold anymore. This has no implications on the liveness-related theorems.

11.3.4 Widening

The widening we implement in the CTGC system for Mercury works as follows: if the size of the structure sharing set exceeds a certain threshold, then every sequence of selectors (or t-selectors) is replaced by the type of the type node that that sequence selects. This operation not only reduces the complexity of the involved selectors, but in most cases also reduces the number of structure sharing pairs.

Example 11.11 *When widening is applied to the structure sharing information in Example 11.5, then the set of abstract structure sharing pairs is reduced to the eight pairs depicted in Figure 11.4. Note that the widening of the structure pair pair* $\left(B^{\overline{(two,1)}} - A^{\overline{(two,1)}}\right)$ *results in the pair* $\left(B^{\overline{\mathbf{T}}^\sharp} - A^{\overline{\mathbf{T}}^\sharp}\right)$ *which is subsumed by the pair* $\left(B^{\overline{\epsilon}^\sharp} - A^{\overline{\epsilon}^\sharp}\right)$.

The structure sharing analysis then continues with the widened set of structure sharing pairs until the size of the set of structure sharing pairs again exceeds the predetermined threshold in which case the widening is repeated.

Finding a good threshold is more of a heuristic matter.

11.3.5 Implementation Issues

Although the operations manipulating t-selectors are more complicated then the ones for manipulating normal selectors, the impact on the analysis time remains limited. In general, most of the time the analysis deals with normal selectors. Widening is only applied in extreme cases, and then the number of structure sharing pairs is reduced drastically. This will be shown in the results of our benchmarks. For our benchmarks we found that a threshold of 500 structure sharing pairs already drastically reduces the analysis time, yet without losing too much of the precision, hence with little impact on the detected structure reuse opportunities.

11.4 Non-local Reuse: Cell Cache

Currently we assumed that every dead data structures may only be reused locally, *i.e.*, within the same procedure in which it is last accessed. This means that we can not detect reuse opportunities where a data structure becomes available for reuse within a procedure p, yet is reused within a procedure q, hence missing quite interesting possibilities of reuse.

Example 11.12 *Consider the following sketch of procedure definitions:*

```
p(...) :- ..., X => f(Y, Z), ...
q(...) :- ..., T <= f(A, B), ...
r(...) :- ..., p(...), q(...), ...
```

If $X^{\overline{\epsilon}}$ becomes available for reuse in p yet p has no possibilities of reusing that data structure, then we would like to be able to reuse $X^{\overline{\epsilon}}$ in another procedure, in this case, ideally in q.

We see three ways to achieve non-local reuses as well. The first and the most difficult is to extend the reuse analysis to explicitly handle non-local reuse. The analysis would have to propagate possible dead data structures and thus become quite complex. It would also require intensive changes in the internal calling convention of procedures within the compiler[6] as the address of the heap cells to be reused would have to be passed between procedures. The second approach is to combine reuse analysis with inlining in such a way that the place where a data structure become dead and the construction unification in which it can be reused end up in the same procedure. The third approach, which is the one we implement, is to *cache* dead data structures. Whenever a data structure, called

[6]The Melbourne Mercury Compiler in our case.

cell in this context, becomes available for reuse unconditionally (thus does not depend on the exact call description of the procedure) and can not be reused locally, we mark it as *cacheable*. The decisions of when these cacheable cells are reused are then taken at run-time. Indeed, the idea is that at run-time the address of the cell as well as its size are recorded in a *cache*, also called a *free list*, a practice that is common in the world of run-time garbage collectors (Wilson 1992). Before each memory allocation the run-time system first checks the cell cache to see if a cell of the correct size is available. If this is the case, it uses that cell instead of allocating a new cell. This operation increases the time taken to allocate a memory cell in the case of the cell cache being empty, and therefore should only be a win if the cell cache occupancy rate is high. Nevertheless, it can avoid new allocations so the overall cost of the run-time garbage collection system should go down due to smaller heap sizes and less frequent need for garbage collection.

11.5 Conclusion

This chapter describes some practical aspects needed for a complete working compile-time garbage collection system. These aspects include the intelligent choice of matching constructions for dead data structures, optimisation of the propagation and representation of the structure sharing, as well as adding a cell cache as a way to try to reuse even those data structures for which no local reuse can be found.

Chapter 12

Benchmarks

In this chapter we detail the implementation of the CTGC system integrated into the Melbourne Mercury compiler and report on the obtained results for a number of small to medium-sized benchmarks.

As we will see, the result is a competitive compile-time garbage collection system, for which some of the benchmarks yield a reduction in memory usage of up to 50%.

12.1 Implementation Details

The structure of the Melbourne Mercury compiler was sketched in Section 3.8.1. Roughly speaking, the different compiler passes are organised into three levels: the high level analyses and transformations based on a high level representation of the source code, called the *High Level Data Structure*, in short HLDS; the low level program code manipulations using the *Low Level Data Structure* representation of the transformed source code; and finally, the actual code generation. The CTGC system is added to the compiler as one of the last high level analyses handling the HLDS representation of the source code. This is needed to guarantee that all semantic checks such as mode-correctness, type-safety and determinism-information are performed. It also allows us to feed back the analysis results obtained from the CTGC system into the HLDS representation using low-level information regarding the structure reuse that is detected. Moreover, none of the subsequent compiler passes may alter the order of the literals within the procedure definitions, which is the case by letting the CTGC system be the last HLDS-manipulating compiler pass.

The implemented CTGC system follows the same structure as the prototype system depicted in Figure 10.2, except that the output is no longer simple annotated source code merely to be used by a human reader, but a carefully annotated

HLDS representation that serves as input for the actual code generation pass of the compiler.

We briefly sketch some of the implementation issues for each of the analysis-steps within the CTGC system.

- *In use information:* Backward use and forward use are pure syntactic properties. These properties are both derived in a separate pass, recorded in the HLDS and therefore serving as pre-annotations to the remaining analysis passes.

  Unlike in our prototype implementation, where backward use information is gathered using an actual analysis, we implement backward use using the simplified approach. As a default we use Instantiation 2 (Section 7.3.2). In Section 12.6 we study the effect of choosing the less precise approach using Instantiation 1.

- *Structure sharing:* We implement the structure sharing analysis following the goal-independent based semantics, using a bottom-up strategy. This goal-independent approach enables the correct analysis of Mercury programs organised into separate modules. Widening can be enabled on a per module basis. The result of the analysis is a collection of exit local descriptions of each of the involved procedures. We have chosen not to pre-annotate the individual program points of interest, as the structure sharing descriptions tend to be big. This means that whenever structure sharing is needed within a specific procedure, a new derivation has to be performed. Given the exit descriptions that are recorded, this does not require a fixpoint process. With this approach, we have traded space for time.

  In our previous prototypes we implemented structure sharing sets using ordered lists making the access and search of particular data structures a complex operation. We alleviated that problem by using a tree based structure[1].

- *Liveness annotation:* The role of the liveness annotation pass is reduced to identifying the deconstruction unifications allowing direct reuse and computing the corresponding (compacted) reuse information tuples. As other unifications are not of interest, they are ignored by the liveness annotation pass. We prefer to compute the required information on demand, hence we postpone the computation of liveness information for procedure calls to the actual moment where that information is needed, *i.e.*, during the derivation

[1] The keys of the tree are the variables of interest, the elements are again trees, this time trees of term selectors, making the access to what a specific data structure is shared with more efficient. The leaves of the latter trees consist of plain lists of data structures. The structure presents some redundancy, as every structure sharing pair can both be accessed by its first data structure as well as by its second data structure.

of indirect reuses. Recall that liveness information depends on structure sharing. We chose not to pre-annotate the code. Therefore we need to partially re-derive structure sharing. Recall that in the presence of the local exit descriptions, this process does not require a fixpoint computation.

Liveness is merely an annotation pass and not an analysis pass. The individual procedures of a module can be handled in any order.

Liveness sets are also represented in a tree based way instead of the ordered lists as we did in our earlier prototypes.

- *Reuse analysis:* Reuse analysis consists of two parts, namely detecting direct reuses — deciding where to reuse the detected dead data structures, and deriving indirect reuses — propagating the reuse information tuples up the call graph.
  - *Direct reuse:* We implement each of the different constraint and selection strategies for assigning construction unifications to dead data structures as developed in Section 11.1.
  - *Indirect reuse:* Indirect reuses are verified and propagated with the assumption that for every procedure at most two versions are created, and that a call to a reuse version is allowed if and only if all the reuse information tuples for that particular call are satisfied.

    In order to verify the reuse information of a called procedure, knowledge about structure sharing and liveness is needed. This means that structure sharing is needed along the way. In general propagating indirect reuses requires program analysis with fixpoint computation. We implement this analysis as a bottom-up process.
- *Interface files:* As a side-effect of the structure sharing and reuse analysis of a given module we create an interface file for each of the analysed modules. The interface file of a module records the exit local structure sharing descriptions of the exported procedures of that module as well as the relevant information concerning the reuse opportunities within each of these exported procedures. The latter consists of the reuse information tuples needed for verifying the safeness of calling the reuse version, and the name of that reuse version of that procedure (usually a variation on the original procedure name). Whether or not cell caching is enabled is a purely internal matter, and therefore not visible in the interface file.
- *Version generation:* Of those procedures for which conditional reuse has been detected, a copy is made and the reuses are incorporated. If some cases of unconditional reuses are detected for a procedure, then these reuses are also enabled in the original version of that procedure. If no unconditional reuses are found, then the original version of the procedure remains untouched.

- *Code generation:* The resulting annotated procedures are compiled into high-level C-code (one of the back-ends of the MMC) taking into account the detected forms of reuse. Local reuse is realised by adding the instructions that explicitly update the adequate memory cells. In the presence of cell caching (see Section 11.4), a deconstruction of an unconditionally dying data structure for which no matching construction unification is found is translated into an instruction placing the corresponding memory cells into the cell cache. With cell caching enabled, each simple memory allocation is replaced by a sequence of operations that first verifies if an adequate memory cell can be found in the cell cache.

12.2 Benchmarks Setting

We evaluate the effectiveness of our CTGC system by comparing memory usage and measuring compilation times. We use toy benchmarks and two real-life programs: a deterministic ray tracing program and a finite domain solver implemented through a number of non-deterministic procedures.

Section 12.3 and Section 12.4 report on results obtained with a CTGC system integrated into version 0.9.1. of the MMC. The system is run on an Intel-Pentium III (600MHz) with 256MB RAM, using Debian Linux 2.3.99, under a usual (small) workload. The results in these sections are published in (Mazur, Ross, Janssens, and Bruynooghe 2001).

Section 12.5 and Section 12.6 present new material obtained with a CTGC system integrated into a more recent version of the MMC w.r.t. the previous experiments. In these sections, the experiments are run on an Intel-Pentium 4 (2.50GHz) with 512MB RAM, using Debian Linux 2.4.20, under a usual (small) workload.

All the reported memory information is obtained using the MMC *memory profiler*. This profiler is part of the Melbourne Mercury compiler. The user program, when compiled with the right option, is augmented to contain low-level instructions that monitor each memory allocation on the heap. After running such an augmented program, the profiler counts the total number of memory words that needed to be allocated during the execution of that program. This count is independent of the interventions of the run-time garbage collector. The allocations can also be viewed for each procedure separately.

Time profiling can be done using a similar *time profiler*. Given the fact that we are not interested in the time consumption of each procedure separately, we perform our timings without this time profiler and simply time the execution of the user program[2]. The timings reported in this chapter are averages of 10 execution runs each time.

[2]We use the user time as returned by the built-in operation *time* of the GNU Bourne-Again SHell, known as *bash*.

module	No Reuse			Reuse			
	C (sec)	M (Word)	R (sec)	C (sec)	M (Word)	m (%)	R (sec)
nrev	1.49	9M	1.51	11.79	6k	-99.9	0.32
qsort	1.40	50M	36.63	11.29	20k	-99.9	27.22
argo_cnters	4.53	3.00M	0.35	16.38	2.60M	-13.3	0.32

Table 12.1: Toy benchmarks. C = compilation time. M = number of allocated words. R = execution time. m = relative reduction in memory usage.

All the benchmarks are compiled using a plain non-optimised setting of the Mercury Standard Library, unless stated otherwise. All the procedures defined in this standard library are compiled without structure reuse as it allows us to focus on the reuses occurring in the actual code of the benchmarks. Nevertheless, for a correct and precise derivation of structure sharing information in our benchmarks we do analyse the procedures of the Mercury Standard Library w.r.t. structure sharing. This results in a set of interface files that is used when analysing the modules of the benchmarks themselves. On the platform based on the Intel Pentium 4 processor as described above, the time needed for installing a version of the compiler with a standard library with structure sharing enabled takes 20 minutes, compared to approximately 10 minutes for installing that same compiler in the same grades with a library without structure sharing information. As these interface files need only be generated once[3], this extra installation cost is acceptable.

12.3 Toy benchmarks

We use the following toy programs: *nrev*, the naive reversal of a list of 3000 integers; *qsort*, sorting a list of 10000 integers using the quicksort algorithm; and *argo_cnters*, a benchmark counting various properties of a data file. Note that the program *argo_cnters* was also used in (Mazur, Janssens, and Bruynooghe 2000) and reported on in Chapter 10. Each of these programs is defined in one single module each.

Table 12.1 shows the compilation time, execution time and memory usage of each of these programs when compiled without structure reuse and with structure reuse. The exact CTGC configuration, *i.e.*, the combination of selection strategy and constraint, has no influence on the reported results.

For each of the benchmarks every possible reuse is detected, yielding the expected savings in memory usage as well as execution time:

[3]In fact, these interface files are derived from scratch for each new grade in which the library is compiled. This is a cost that can easily be avoided.

- Within *nrev* the CTGC system is able to recover every list cell deconstructed.
- The partitioning procedure used in *qsort* does not need to allocate any new memory as everything can be reused locally.
- For the *argo_cnters* benchmark, reuse is also performed successfully: the data structure recording the properties of the file being updated in place. The difference in timing for this benchmark is statistically insignificant as most of the execution time is due to input/output operations.

12.4 Ray Tracer, Take I

Next to small benchmarks, we found it important to evaluate the system on a real-life program. The main goal of this experiment is to study the effect of the choice of the exact CTGC configuration, *i.e.*, selection strategy and constraint, on the amount of structure reuse obtained.

The first program we use is a ray tracer program developed for the ICFP'2000 programming contest (Morrisett and Reppy 2000) where it ended up fourth. This program transforms a given scene description into a rendered image. It is a CPU- and memory-intensive process, and therefore an ideal candidate for our CTGC system to be tested on. A complete description of this program can be found at (The Mercury Team 2000).

12.4.1 Description

The program consists of 20 modules (4000 lines of code, declarations included), containing only deterministic predicates and functions. All modules can be compiled without widening, except for one: *peephole*. This module manipulates complex terms and generates up to 11K aliases. Without type widening, the compilation of *peephole* takes 160 minutes. With type widening (using a threshold of 500 structure sharing pairs), it only takes 40 seconds. The compilation of the whole program with CTGC (and widening) takes 5 minutes, compared to 1 minute for a normal compilation. As some of these modules are mutually dependent, the technique of manually iterating the compilation is used to obtain better results. For this benchmark, the compilation needs to be repeated three times to reach a fixpoint (for a total time of 15 minutes). Each time every module is recompiled. In a smart compilation environment, most of the recompilations can be avoided.

12.4.2 Results

Table 12.2 compares the use of memory and time resources of the ray tracer compiled without any form of CTGC with the resources needed for versions of the ray tracer compiled with different CTGC configurations. The figures shown in that table represent the total memory and time usage of rendering a set of 27

different scene descriptions (ranging from simple scenes to more complex ones) with each of the ray tracer versions. The first row shows the figures obtained for the ray tracer without any CTGC, rows 1 to 8 give the results for versions of the ray tracer without cell caching, while rows 10 to 17 relate to versions of the ray tracer with the option of cell caching enabled. In order to measure the effect of structure reuse in the Mercury Standard Library procedures we compiled one version of the ray tracer using a library with CTGC. The results of this ray tracer are shown in row 9. Note that at the time these figures were compiled, the graph based deconstruction-construction allocation had yet to be incorporated into the CTGC system, therefore figures for that new allocation scheme are not available here[4].

12.4.3 Observations

We make the following observations:

- Using the matching arities (*match*) or label-preserving (*same cons*) constraints, up to 24% memory can be saved globally. For some scene descriptions, this can go up to 30%. There is also a noticeable speedup (14%).
- Using almost matching arities within a distance of one (*within 1*) or two (*within 2*), much less memory is saved (only 10%) with hardly any speedup. The bad memory usage is not surprising as none of the selection strategies takes into account the correspondence of the arities between a new cell and the available dead cells. This problem can definitely be alleviated using the graph based approach, as will be confirmed later. The bad timings are also explicable: with non-matching arities, reuse leaves space-leaks which can not immediately be detected by the current run-time garbage collector, hence the garbage collector will be called more often.
- We have also experimented with configurations using *random* as a selection rule. Combined with the selection constraints, results similar to their *lifo* counterparts are obtained. The difference in amount of memory reused manifests itself mostly in the presence of disjunctions within the definition of the procedure. E.g. `X => f(...), (...Y <=f(...) ; ...), Z <=f(...)`. As the first branch of the disjunction might not always be executed, it is more interesting to allow `Z` to reuse `X` than `Y`.
- Row 9 shows the results of a ray tracer compiled using a version of the Mercury Standard Library *with* CTGC. There is hardly any difference with Row 1, where a library is used without structure reuse enabled. The limited impact is simply due to the limited use of the library procedures by the ray tracer program.

[4]Figures for this new strategy are given in the following two sections.

	Configuration			Memory (kWord)	(%)	Time (sec)	(%)
0	no CTGC			1024795.51	-	362.31	-
1	lifo	match		776707.92	-24.21	311.85	-13.93
2	lifo	same cons		791742.06	-22.74	313.57	-13.45
3	lifo	within 1		916642.90	-10.55	361.84	-0.13
4	lifo	within 2		917847.97	-10.44	359.90	-0.67
5	random	match		780838.58	-23.81	310.75	-14.23
6	random	same cons		795872.67	-22.34	312.70	-13.69
7	random	within 1		920764.26	-10.15	359.14	-0.87
8	random	within 2		921969.35	-10.03	355.08	-2.00
9	lifo	match	libs	775607.04	-24.32	320.32	-11.59
10	lifo	match	cc	513901.37	**-49.85**	301.66	-16.74
11	lifo	same cons	cc	542626.80	**-47.05**	304.20	-16.04
12	lifo	within 1	cc	845603.55	-17.49	375.79	3.72
13	lifo	within 2	cc	864722.90	-15.62	370.49	2.26
14	random	match	cc	518032.04	**-49.45**	299.45	-17.35
15	random	same cons	cc	546757.48	**-46.65**	302.79	-16.43
16	random	within 1	cc	849724.90	-17.08	363.90	0.44
17	random	within 2	cc	868844.29	-15.22	391.68	8.11

Table 12.2: ICFP ray tracer results using different CTGC configurations.

- The *cc*-entries of Table 12.2 (Rows 10-17) show the results of CTGC-configurations combined with the cell cache technique. Compared to the basic CTGC-configurations, cell caching always increases memory savings, going up to 49% (for some scenes even 70%). In the case of label-preserving or matching arities constraints, we obtain a slightly worse execution time. On the other hand, using almost matching arities combined with cell caching increases the execution time.

12.4.4 Conclusion

For all of the CTGC configurations, a remarkable amount of memory reuse can be achieved. The best results are obtained using the lifo selection strategy, in combination with matching arities or label-preserving constraints. The liberal constraints *within 1* and *within 2* yield poor results. In the following section we will see that these poor results can partially be alleviated when used in combination with a better selection strategy.

Cell caching increases memory savings for all of the basic CTGC configurations, yet may have a negative effect on the execution time of the optimised ray tracer.

12.5 Ray Tracer, Take II

More recently we repeated the ray tracer experiments to study the new graph based approach for selecting matching deconstruction/construction unifications. Moreover, with faster hardware, we could also experiment more easily with type widening and its effect on the obtained structure reuses.

12.5.1 Description

The main changes between this setting for the ICFP ray tracer, and the setting used above are:

- Graph based allocation of deconstructed dead cells for new to be constructed cells is now present in the CTGC system;
- Some annoying bugs are removed, especially one bug where the reuse information detected in else-branches of if-then-else goals was erroneously discarded, hence reuse in the else-branches was never possible.
- The CTGC is embedded into a separate branch of the development tree of the MMC. Once in a while this branch needs to be synchronised with the main branch in order to profit from the changes and fixes of the main compiler. While the CTGC used above was last synchronised with the main branch in September 2001, the CTGC used here was upgraded in August 2002;
- And finally, the experiments are run on an Intel-Pentium 4 (2.50GHz) processor with 512MB RAM, using Debian Linux 2.4.20.

Note that for completeness sake we also repeated the toy benchmarks for this new platform. All yielded similar results as previously[5] which is why we do not report on these benchmarks again.

12.5.2 Results

Table 12.3 lists the memory usage of the different CTGC configurations with which the ICFP ray tracer is compiled. The execution times are shown in Table 12.4. Compared to Table 12.2 we now have additional entries which report on the graph based selection strategy. No timings are shown.

Detailed memory usage as well as timings for each of the sceneries rendered by the ray tracer are given in Appendix B.

[5]The *argo_cnters* benchmarks manifests a slow down when compiled with structure reuse enabled. We refer the reader to Section 12.5.5 where a similar observation for the ray tracer is discussed.

	Configuration			Memory (kWord)	(%)
0	no CTGC			1024354.9	-
1	graph	match		762167.72	-25.60
2	graph	same cons		777192.11	-24.13
3	graph	within 1		739539.99	-27.80
4	graph	within 2		739539.99	-27.80
5	lifo	match		765195.52	-25.30
6	lifo	same cons		780219.92	-23.83
7	lifo	within 1		757462.96	-26.05
8	lifo	within 2		950359.45	-7.22
9	random	match		769326.21	-24.90
10	random	same cons		784350.61	-23.43
11	random	within 1		761593.35	-25.65
12	random	within 2		957421.59	-6.53
13	graph	match	cc	514845.18	**-49.74**
14	graph	same cons	cc	525640.75	**-48.69**
15	graph	within 1	cc	683984.62	-33.23
16	graph	within 2	cc	683984.62	-33.23
17	lifo	match	cc	517873.02	**-49.44**
18	lifo	same cons	cc	528668.61	**-48.39**
19	lifo	within 1	cc	701895.34	-31.48
20	lifo	within 2	cc	894791.85	-12.65
21	random	match	cc	522003.66	**-49.04**
22	random	same cons	cc	532799.23	**-47.99**
23	random	within 1	cc	706025.77	-31.08
24	random	within 2	cc	901854.01	-11.96

Table 12.3: Memory usage of the ICFP ray tracer configured with different CTGC settings.

12.5.3 Basic Observations

Comparing the CTGC configuration of Table 12.3 with the corresponding configuration of Table 12.2, we see that the memory usage is slightly better than previously. This was expected as due to the bug-fix that now also enables reuse in the else-branches of if-then-else constructs, more opportunities for reuse are detected.

We observe that for the *within 1* configurations, reuse is truly better, yet for the *within 2* constraint, we obtain much less reuse than in the old setting. This can be explained by the larger set of dead cells that is now detected and the blind allocation schemes that are used (*lifo* and *random*): indeed, the choice of the dead cell for creating a new cell can sometimes be unfortunate as it imposes reuse conditions that can not be met by all of the callers of the involved procedure, hence limiting the overall reuse.

Table 12.3 allows us to compare the new graph based allocation scheme with the basic schemes used before. As could be expected, the graph based configurations (the *graph* entries in the table) yield better reuse results than the lifo and random configurations. The superiority of the graph based approach can especially be observed for the liberal reuse constraints *within 1* and *within 2* (compared to these constraints in the *lifo* and *random* settings) where both give the same amount of reuse.

12.5.4 Cell Caching

Just as in our elder setting, cell caching increases the overall memory reuse opportunities of the program (c.f. the *cc* entries of Table 12.3). The increase is especially noteworthy for the CTGC configurations with the label preserving (*same cons*) and matching arities constraints (*match*): without cell caching we obtain approximately 25% of memory savings, while with cell caching, this saving can reach almost 50%.

The effect of cell caching is much less spectacular for the more liberal selection constraints, *i.e.*, *within 1* and *within 2*. Recall that cell caching is only enabled for cells that die *unconditionally*, hence, differences in number of derived reuse conditions for the procedures in the different CTGC settings can not explain the difference in reuse. Therefore, the only way we can explain the limited influence of cell caching for the *within 1* and *within 2* configurations is that in these settings more local reuses are detected, reuses that may also involve data structures that die unconditionally. In other configurations these data structures will die unconditionally too, yet no matching construction will be found. With cell caching enabled, this means that these data structures will be placed into the cell cache and made available for global reuse. It suffices that the local reuses detected with the *within 1* and *within 2* configurations cover less execution paths then the global reuses of these same data structures in the other configurations, to explain the

lesser impact of cell caching on the global memory saving for the *within 1* and *within 2* configurations. The current implementation of the CTGC system does not allow a clear insight into this specific phenomenon, yet we believe that if it would, it would only confirm the above expressed intuition.

Note that the same behaviour could already be observed in Table 12.2.

12.5.5 Time Profiling

We expected the same positive results for the time profiling as for the memory profiling. Yet the opposite happens: while previously we had configurations where the run-time could be reduced by almost 15%, here we have an increase of almost that same amount, and hardly any configuration yielding any kind of run-time decrease. The timings and also the number of run-time garbage collection interventions are shown in Table 12.4. A closer look into the timings (see Appendix B for the complete details of the timings for each of the sceneries with which the ray tracer was benchmarked) reveals that sometimes large memory reuse comes along with large run-time reductions (e.g. scenery `cylinder.gml` in the *graph-match* configuration uses 27% less memory than in the base case, and takes almost 11% less time to render), yet even in the same configuration, examples can be found where large memory reuse come with an increase in execution time (scenery `mtest7.gml` in the *graph-match* configuration uses 23% less memory, yet takes more than 12% of time to render). While these results left us mystified in the first place, the only cause for that total change of behaviour we could identify is the different run-time garbage collectors used in the both CTGC systems. Indeed, between the elder version and the current version, the mercury run-time system was upgraded to a new version of the Boehm garbage collector[6]. We think that the effects of our local reuses probably counteract the locality principles of the run-time garbage collector (RTGC). Indeed, the run-time garbage collector is not aware of the explicit memory reuses introduced by our CTGC system. This means that the set of heap cells considered live from the point of view of the run-time garbage collector can be bigger than that set of heap cells considered live without the allocations introduced by the CTGC system. With this larger memory footprint, memory accesses become more expensive, and the number of run-time garbage collection interventions increases. The last effect is confirmed by the figures shown in Table 12.4.

This experiment makes clear how the CTGC can counteract the RTGC. It is definitely worth a closer look to see how the CTGC and RTGC systems can be tuned for a better cooperation.

[6]While the elder version uses `gc6.0alpha6`, the upgraded version depends on `gc6.1alpha5`.

	Configuration			Time (sec)	Time (%)	GC (times)	GC (%)
0	no CTGC			87.53	-	77924	-
1	graph	match		87.97	0.50	89063	14.29
2	graph	same cons		88.28	0.86	83824	7.57
3	graph	within 1		86.09	-1.65	84102	7.93
4	graph	within 2		86.10	-1.63	84102	7.93
5	lifo	match		87.37	-0.18	82103	5.36
6	lifo	same cons		87.17	-0.41	83624	7.31
7	lifo	within 1		87.02	-0.58	81864	5.06
8	lifo	within 2		93.24	6.52	91794	17.80
9	random	match		87.61	0.09	82597	6.00
10	random	same cons		86.87	-0.75	83855	7.61
11	random	within 1		90.32	3.19	81804	4.98
12	random	within 2		94.84	8.35	91842	17.86
13	graph	match	cc	93.93	7.31	77375	-0.70
14	graph	same cons	cc	94.42	7.87	75499	-3.11
15	graph	within 1	cc	95.33	8.91	84166	8.01
16	graph	within 2	cc	95.42	9.01	84166	8.01
17	lifo	match	cc	93.21	6.49	77346	-0.74
18	lifo	same cons	cc	95.15	8.71	75529	-3.07
19	lifo	within 1	cc	96.17	9.87	84436	8.36
20	lifo	within 2	cc	102.34	16.92	94180	20.86
21	random	match	cc	93.10	6.36	77545	-0.49
22	random	same cons	cc	94.95	8.48	75681	-2.88
23	random	within 1	cc	95.46	9.06	81049	4.01
24	random	within 2	cc	103.05	17.73	93162	19.55

Table 12.4: Time profiling of the ICFP ray tracer configured with different CTGC settings. The last columns compare the number of run-time garbage collection interventions during the execution of the ray tracer of the optimised ray tracer w.r.t. the *no reuse* version.

12.5.6 Type Widening

In the early experiments with the ICFP ray tracer, compiling that ray tracer without type widening was almost impossible, some of the modules requiring up to two hours of compilation and analysis time. Taking into account that the occurrence of programming errors in the CTGC system was still considerable at that stage, and therefore, the compilation of the ray tracer had to be repeated very often, type widening was a bare necessity.

With more modern hardware, the compilation and analysis time of those same modules is now reduced to a couple of minutes: the full compilation of the ray tracer, including the recompilation to take into account the mutual dependence of the modules now takes 15 minutes without any widening, compared to 5 minutes with widening. This new setting, together with a more stable CTGC system, enables us to study the effect of type widening on the degree of reuse that can be detected by the CTGC system. We can now compile the ray tracer with configurations where type widening is enabled at different thresholds, or even not at all.

We compile the ray tracer using three different configurations: (1) a ray tracer without any type widening, (2) a ray tracer in which only the bottleneck module *peephole* is compiled with type widening (threshold 500) — this corresponds to the default compilation setting used in the previous experiments — and (3) a ray tracer in which *every* module is compiled with type widening using a threshold of 50 structure sharing pairs. Given the low threshold, the last configuration forces type widening at each step in the analysis and can thus be seen as a fair simulation of a pure type based approach. In such an approach all term and type selectors are replaced by a description of the type that these selectors designate.

Astonishingly, type widening seems not to have any effect at all on the detected opportunities for reuse as all three configurations yield exactly the same final executable program, hence exactly the same reuses are detected. This is an important observation in favour of the type based approaches as it shows that the use of the complex term and type selectors can be replaced by the more modern fully type based approaches while still guaranteeing precise analysis results.

See Section 12.7 for a further discussion on this issue.

12.5.7 Structure Reuse in the Mercury Standard Library

We also study a version of the ICFP ray tracer using a version of the Mercury Standard Library in which structure reuse is enabled. Without giving the full details of the figures, the general tendency is that less memory can be reused compared to the ray tracer configurations compiled with a library without structure reuse.

A closer look into the code reveals that the conditional reuses appearing in the library procedures propagate into new conditional reuses in the ray tracer

procedures. As a result these procedures become characterised with a set of reuse conditions which become harder to meet by the callers of these procedures. Given our policy of only producing two versions for each of the procedures — one with full reuse, another with only unconditional reuse — calls to procedures with full potential of reuses are now replaced by procedures with only the unconditional reuses, hence, yielding an overall decrease in memory reuse opportunities.

This again illustrates that the policy of only creating two versions for each of the procedures in which some reuse is detected can be too limiting for obtaining good memory reuse behaviour.

Note that in the old ICFP ray tracer setting we did not observe a comparable deterioration of the structure reuse results. The main reason is that in that old setting, reuses in else-branches of the if-then-else goals were erroneously discarded, hence less reuses were detected, yielding less reuse conditions.

12.5.8 Conclusion

In this new setting we have experimented with the graph based selection strategy. This selection strategy outperforms any of the more naive strategies used earlier, *i.e.*, *lifo* and *random*. While liberalising the selection constraint allowing reuses even when the arities differ in arity by one (*widen 1*) or by two (*widen 2*) has a negative effect on memory reuse in the *lifo* and *random* settings, using the graph based selection strategy, overall memory saving can be increased. Nevertheless, the *widen 1* and *widen 2* constraints are not recommended in the presence of cell caching.

Using this updated CTGC system that relies on a new run-time garbage collector, we do not obtain the same gain in execution time as we did in our earlier experiments. This reveals that a good cooperation between the CTGC system and the RTGC system is essential to increase the effect of CTGC on execution times.

As a final conclusion from this experiment, we observe that for this particular benchmark type widening has no effect on the amount of reuses detected by the CTGC system.

12.6 Finite Domain Solver

Next to studying the CTGC system for fully deterministic Mercury code, we also want to study its potential in a more non-deterministic setting. A good benchmark for this is the Finite Domain Solver ROPE (Vandecasteele 1999a), implemented in Mercury, the resulting implementation being called MROPE II (Vandecasteele, Demoen, and Janssens 2000; Vandecasteele 1999b).

The main goal of the experiment of compiling the MROPE II library and its benchmarks using the CTGC system is to evaluate the effect of non-deterministic code on the obtainable structure reuse. We also want to relate this effect to the

choice of one of the two proposed models for the simplified backward use derivations. The effect of choosing the right CTGC configuration is of less importance here.

12.6.1 Description

MROPE II is written as a stand-alone library providing the necessary primitives to express constraints, collect them in a constraint store, and finding solutions for the obtained constraint systems. The Mercury code of the MROPE II library presents the following characteristics:

lines of code	4000
# modules	13
# procedures	239
# exported procedures	133
# non-deterministic procedures	6
# non-deterministic exported procedures	5

We slightly modify this library for the purpose of our analyses:

- The original library contains two modules that are not only mutually dependent, but they also contain mutually recursive procedures each defined in a different module. In such a setting, even the repetitive application of the analysis will never yield good results for the structure sharing. As such practices are considered as bad programming habits, we simply merge these two modules.
- The library heavily relies on the Standard Mercury Library module *bt_array* (defining a backtrackable array). While we still want to compile the benchmark with a Mercury Standard Library that is not optimised w.r.t. structure reuse, we do recognise that not optimising this module severely limits the effects of our CTGC system. We therefore explicitly include this library module into the the MROPE II library in order to allow structure reuse to be detected in that module too. This module has been taken into account in the characteristics summarised above.
- The module *non_logical_io* contains procedures for performing I/O implemented using side effects. As a result, these procedures can be used in a non-deterministic context yet their use breaks the pure declarative nature of the code and therefore limits the compiler in its possibilities of reordering the code. The use of these procedures has no actual influence on the CTGC system, except that, in order to obtain satisfying results w.r.t. precision, we explicitly add annotations that declare that none of these procedures add any extra structure sharing (c.f. Page 251).

The non-deterministic procedures defined in the MROPE II library are the procedures searching for solutions of the constraint store.

	loc	#proc	#nondet
`queens`	66	5	0
`bridge`	450	22	10
`bridge1`	450	23	11
`suudoku`	230	18	2
`perfect`	150	14	1
`eq10`	80	5	1

Table 12.5: Technical details of the benchmarks used for the MROPE II library (loc = lines of code, #proc = number of procedures, #nondet = number of non-deterministic procedures)

The benchmarks used for the MROPE II library include some typical CLP (Jaffar and Maher 1994; Stuckey and Marriott 1998) problems: *queens* — solving the queens problem, *bridge* (Van Hentenryck 1989), *bridge1* (a variation on *bridge*), and *suudoku* – solving a Japanese puzzle. Other smaller benchmarks that we also used as a benchmark are *perfect* (generating so called *perfect* squares), and *eq10* (solving a set of mathematical equations).

Each of these benchmarks is implemented as a separate module. The technical details of these modules are listed in Table 12.5. Note that each of these modules only exports one predicate/procedure, namely the `main/2` procedure.

We compile the library and its benchmarks in one single CTGC configuration: we use the graph based allocation scheme, restricting reuses to data structures having the same constructor. Initially we restricted the data structures to be matching, yet it appears that too much reuse is detected to be useful due to the limited number of versions that we generate per procedure, hence our more strict constraint. Cell caching is enabled, although hardly any of the reuses in this library can be accounted to global reuses. As always, we use a Mercury Standard Library that is not optimised w.r.t. structure reuse.

12.6.2 Results

We compile the finite domain solver and its benchmarks with the two presented simple instantiations of the backward use information. We obtain the memory usage results shown in Table 12.6. The column labelled NoR details the memory usage (in kilo-Words) of the benchmarks compiled without CTGC. The entries R_1 and R_2 refer to the absolute memory usage for the benchmarks compiled using bu^1, and bu^2 resp. (c.f. Section 7.3.2). The relative memory improvement due to CTGC with either of the backward use definitions, compared to no CTGC is represented in the columns labelled r_1, resp. r_2. More specifically: $r_1 = (NoR - R_1)/NoR * 100$, and $r_2 = (NoR - R_2)/NoR * 100$. The columns labelled cr_1 and

	NoR (kW)	R_1 (kW)	R_2 (kW)	r_1 (%)	cr_1 (%)	r_2 (%)	cr_2 (%)
`bridge`	12695.19	12694.19	12681.26	-0.01	-0.01	-0.11	-0.11
`bridge1`	48225.34	48225.27	48212.34	-0.00	-0.00	-0.03	-0.03
`eq10`	1042.67	951.65	951.45	-8.73	-16.77	-8.75	-16.81
`perfect`	2890.35	2889.20	2889.20	-0.04	-0.05	-0.04	-0.05
`perfect*`	2890.35	2889.20	1416.22	-0.04	-0.05	**-51.00**	**-61.67**
`queens` (8)	4169.48	4022.61	4022.41	-3.52	-4.00	-3.53	-4.01
`suudoku`	9810.00	9808.62	7775.51	-0.01	-0.01	-20.74	-21.85

Table 12.6: Memory usage of the MROPE II benchmarks where NoR = No Reuse, in kilo-Word; R = Reuse, in kilo-Word; r = relative decrease in memory usage (*i.e.*, $(NoR - R)/NoR$), in percent; cr = *corrected* relative decrease in memory usage, in percent.

cr_2 is a *corrected* relative memory improvement. Indeed, a closer look into the workings of the finite domain solver reveals that before any constraint can be added to the constraint store, this constraint store is initialised. This initialisation consists of creating an array of size 100000. This represents a fixed cost of 500 kilo-Words, which, for benchmarks of small or even medium size is not negligible. The *corrected* entries therefore remove that fixed cost from the absolute memory usage before computing the relative memory improvement. Thus, $cr_1 = (NoR - R_2)/(NoR - 500) * 100$, and $cr_2 = (NoR - R_2)/(NoR - 500) * 100$.

12.6.3 Overall Reuse

Overall, we see that even in the presence of non-deterministic code, *some* form of memory reuse is still possible. As could be expected, using the second instantiation of the backward use definition yields better results. This can easily be explained by the fact that less variables are considered to be in backward use, and therefore, potentially, more data structures can be detected to be dead.

12.6.4 Clean CLP-formulation

Taking a closer look into the definition of the constraint problems, we see that those constraint problems that clearly separate the phase of defining the constraints from the phase of actually solving these constraints yield better reuse results than those problems in which that separation is less clear. The lack of that separation is for example the case in the *bridge* problem and even worse, in *bridge1*.

Two aspects are involved in this. First, deterministic code remains a more suitable playground for structure reuse than non-deterministic code. As defining

constraints is a pure deterministic matter, it can be expected that optimising the construction of these constraints is usually easier than the non-deterministic process of solving them. The second aspect is related to the fact that both building constraints, and solving constraints, are expressed with procedures manipulating the so called *constraint store*. Each of these procedures takes as input one state of that constraint store, and produce as output a new state of that store. Obviously, the two states will be related by means of structure sharing. As long as the procedures are deterministic, destructive updates by means of structure reuse can be allowed. Yet, at the first occurrence of a non-deterministic procedure handling that store, all the subsequent procedures will not be allowed to reuse parts of that store as we must guarantee that that first non-deterministic procedure call continues to have access to exactly the same input store on each of the possible backtrackings within that procedure. The following sketch of code illustrates these aspects:

Example 12.1 *Let us assume that the MROPE II library defines the (abstract) types* constraint *and* store*, and two procedures declared as follows:*

```
:- pred addConstraint(constraint,store,store).
:- mode addConstraint(in,in,out) is det.
:- pred solve(store,store).
:- mode solve(in,out) is nondet.
```

Let us assume that the CTGC system has detected possibilities of reuse within each of these procedures, and has therefore generated the adequate reuse versions.

Now consider the sketch of a first constraint problem, expressed through a procedure called problemA*:*

```
problemA :-
        ..., % some deterministic code ...
        addConstraint(C1,S1,S2),
        addConstraint(C2,S2,S3),
        addConstraint(C3,S3,S4),
        solve(S4,S5),
        solve(S5,S6),
        ...,
```

and a second constraint problem, defined using a procedure problemB*:*

```
problemB :-
        ..., % some deterministic code ...
        addConstraint(C1,S1,S2),
        solve(S2,S3),
        addConstraint(C2,S3,S4),
        solve(S4,S5),
        addConstraint(C3,S5,S6),
        ...,
```

In problemA, the process remains deterministic up to the first call to solve. This means that the three calls to addConstraint can probably be optimised and therefore replaced by the adequate reuse versions. The first call to solve might also allow some form of reuse as that call has the last and unique access to the store represented by variable S4. Yet given the fact that S4 must be kept intact, and that S5 will probably have some sharing with S4, the literal solve(S5,S6) will not be able to reuse any parts of S5. Hence, reuse in procedure problemA is possible in the three calls to addConstraint and in the first non-deterministic call, in this case, the first occurrence of solve.

If we repeat this reasoning for problemB, where non-deterministic calls are intermixed with deterministic calls, we can see that structure reuse can only be allowed in the first occurrence of addConstraint and in the first occurrence of solve. All subsequent literals deal with a store that has some form of structure sharing with the store used by the first call to solve, and therefore no structure reuse of the store is possible in these literals.

Another common practice in describing CLP-problems is to write the addition of constraints into a constraint store as a non-deterministic disjunction itself:

```
problemB :-
        ..., % some deterministic code ...
        ( addConstraint(C1,S1,S2)
        ; addConstraint(C2,S1,S2)
        ),
        addConstraint(C2,S2,S3),
        solve(S3,S4),
        ...,
```

This reflects the idea that one does not know which particular constraint yields the desired end result. In that setting, the non-deterministic disjunction adds the input store to the set of backward use variables, limiting the possibilities of further reuse within the constraint store just like described for procedure problemB above.

Relating the previous example to the benchmarks of the MROPE II system, we see that the CLP problems *bridge* and *bridge1* follow the scheme of problemB in our example, while most of the other benchmarks resemble more the problemA-procedure.

For such examples we may expect a slightly better structure reuse behaviour using an analysis based approach to backward use structures, yet even with this increased precision, the effect of CTGC will be limited simply because the problems themselves do not leave much room for CTGC to take action. With programs where the data structures handled by the non-deterministic procedures are less connected via structure sharing the CTGC system can probably achieve much better results.

12.6.5 Limiting the Reuse Opportunities

With benchmark *perfect* we again encounter the problem that the CTGC system detects too many opportunities of reuse for some of the procedures, therefore obtaining too harsh reuse conditions, hence limiting the overall memory reduction. In this case, it may be useful to allow the programmer to use a pragma enabling her/him to tell the compiler that reuse should mainly be focused on the constraint store which is the central data structure for constraint solving problems, and not be allowed for other structures that are used less consistently. In the precise case of *perfect*, some procedures not only contain calls to some of the well-optimiseable procedures of the MROPE II library, but also try to locally reuse simple list(T) cells.

We manually change these procedures in such a way that reuse of these list-cells can not be detected anymore. This results in a new version of the *perfect* module, listed in Table 12.6 as *perfect**. While without that modification, hardly 0.05% of the memory usage is saved, we now have a dramatic memory reduction of up to 61%.

12.6.6 Deterministic Versus Non-deterministic code

We can observe that while structure reuse is to some extent also detected in non-deterministic procedures, the main target for the CTGC system remains the deterministic part of a program. Therefore, for those problems in which the deterministic part is more important than the actual search, large memory savings can be expected and are indeed observed. On the other hand, for problems that are dominated by the non deterministic search, the memory savings are much less. This is the case for the *queens* benchmark. A closer look into the memory profiling of that benchmark shows that the memory use of its deterministic part is reduced by up to 88% when compiled with CTGC enabled, while only 0.06% can be saved in the actual search. Solving the *queens*-problem with six queens will therefore reveal a larger memory saving using CTGC then solving that problem with eight queens.

12.6.7 Conclusion

The goal of studying the MROPE II library and its benchmarks in the context of the CTGC system was to evaluate the effect of non-deterministic code on the amount of structure reuse that can be obtained.

We observe that even in the presence of non-deterministic code, our CTGC system is still able to detect local reuses. This means that the approximation of the backward use information using Instantiation 2 (Section 7.3.2) can be considered as sufficiently precise for this benchmark. Nevertheless, structure reuse is mainly effective in deterministic procedures. If the non-deterministic searches become

relatively more important than the deterministic parts of the program, less overall memory saving can be expected.

In some situations, the CTGC system detects too many reuse opportunities for the optimised procedures to be usable. This phenomenon was also observed in the previous experiments, yet here we were able to clearly identify the over-optimised procedures. By rewriting them, thus forcing the system not to recognise some of the reuses, the overall memory saving could be dramatically increased.

12.7 Discussion and Further Improvements

The study of the benchmarks, whether small or medium-sized, has learnt us that the memory usage of programs can be greatly reduced by means of program analysis enabling local structure reuse or cell caching. However, some open issues remain. We discuss these problems, possibly presenting interesting alternatives or guidelines for future work and future experiments.

Type Widening versus Type Based Analysis The experiments performed with the more recent CTGC implementation made the study of the effect of type widening feasible on a larger scale. As it appears, the loss of precision due to type widening is limited. This illustrates the potential for the so called type based analyses where the use of the archaic and complex term and type selectors is replaced by types (Bruynooghe, Codish, Gallagher, Genaim, and Vanhoof 2003; Bruynooghe, Codish, Genaim, and Vanhoof 2002; Codish, Genaim, Bruynooghe, Gallagher, and Vanhoof 2003; Lagoon, Mesnard, and Stuckey 2003; Lagoon and Stuckey 2001), Clearly, type widening allows for faster analyses, hence, type based analyses can in general be expected to be faster than term-selector based analyses.

Still, further experiments would be helpful in really identifying the impact of type based analyses, and the obtained precision.

On a more formal basis, the only adaptation required for adapting the current CTGC system to using types is to adapt the definition of the abstract data structure. The adaptation of all depending domains follows naturally.

Structure Sharing: Space versus Time In the theoretical foundation of the implemented CTGC system we describe that liveness information is derived directly from the structure sharing annotations that we assume to be derived prior to the liveness analysis pass. However, in the current CTGC implementation, we have traded space for time, and re-derive structure sharing at the moment liveness information is computed, using only the goal-independent exit-descriptions of the analysed procedures.

Given the fact that computing local structure sharing information can be a complex operation (given the complexity of the alternating closure operation), it can be interesting to study the behaviour of the CTGC system in case the local structure sharing is recorded as program point annotations. Obviously, only deconstruction unifications and procedure calls should be annotated.

Modularisation: Intelligent Compilation Process In our experiments, especially in the larger benchmarks, recompilation of the modules was required to correctly deal with mutually dependent modules. In the case of the ICFP ray tracer, the program needed to be compiled three times before reaching a fixpoint, for the finite domain solver, up to five passes were required. Clearly, a more modern approach is needed to handle the intermodule dependencies in such situations. Adapting the CTGC to the new compilation scheme according to (Bueno, García de la Banda, Hermenegildo, Marriott, Puebla, and Stuckey 2001) becomes mandatory. An interesting starting point for adapting our analyses is (Nethercote 2001) studying program analysis in the context of HAL (Demoen, García de la Banda, Harvey, Marriott, and Stuckey 1999).

Versioning As illustrated by our benchmarks, the CTGC system easily detects *too many* opportunities for reuse. While this illustrates the strength of the underlying analyses, it also illustrates the weakness of our versioning heuristic: too many opportunities can mean too harsh safeness conditions, hence limiting the overall actual memory reuse.

A first simple solution to this problem is to give the programmer a tool allowing her/him to express his interest (or lack thereof) of reusing specific terms. In the constraint problem *perfect* we drastically augmented the overall memory reuse by luring the compiler into believing that some of the terms are unsafe to be reused. A simple pragma telling the compiler not to reuse terms of that specific type would have been easier.

While many different heuristics or new language constructs can be found or invented, a more interesting and fundamental approach is to consider the versioning problem as a new analysis domain in its own right. In Chapter 13 we give a first tentative formulation for deriving so called optimisation properties, in our view a mandatory stepping stone towards a formal approach to deriving and studying the different optimised versions that can be obtained for each single procedure (or other language entity to be optimised) defined in a program.

CTGC meets RTGC In our first experiments with the ICFP ray tracer, we obtained speed-ups of up to 10% (and for particular sceneries we even reached figures of up to 17%). This was extraordinary, and made us feel euphoric about the potential of compile-time garbage collection. To some extent we can say that luckily, the second series of experiments did not yield these same speed-ups. This

allowed us to identify the fact that speed-ups can only be obtained if the memory reuse possibilities identified by the CTGC system stroke with the locality principles of the run-time system and its garbage collector. This is an interesting research topic on its own, and we believe that by fine-tuning the CTGC and RTGC systems for a good inter-operation, real speed-ups can certainly be guaranteed.

Higher Order Language Features Currently, we ignored the aspect of higher-order programming as if it is not part of the language. Yet even in our ray tracer benchmark, higher order calls are present. Indeed, the ray tracer relies on basic exception-handling facilities which are implemented as higher-order calls. It also uses the classic `map`, `foldl`, `filter` procedures defined for manipulating list structures.

As such, higher-order calls are not handled at all by our CTGC system. Yet it appears that all the higher-order elements in the ray tracer are specialised into plain first-order predicates. The code that the CTGC system therefore handles is a simple first-order logic program, yielding the interesting results discussed earlier.

This seems to suggest that in the presence of a good performing higher-order specialising compiler, there is no real need to extend the CTGC system to cope with higher-order language features itself. Moreover, such an extension is far from trivial. We refer the reader to (Vanhoof 2001) where first-order binding-time analysis for first order Mercury is adapted for higher-order language constructs.

The Ultimate Benchmark As for now, only relatively small benchmarks are studied for the CTGC system. This was deliberate as these benchmarks were used to debug the system, to study the behaviour of the system, to study the missed reuse opportunities, in short: the programs needed to be small enough to be able to verify the results. In the near future, we hope to incorporate the CTGC system into the main branch of the MMC[7] in order to obtain a stable and reliable product in which case we hope to be able to analyse the ultimate benchmark, which is the Melbourne Mercury Compiler itself. Indeed, this compiler is mostly written in Mercury. As previously exposed, there are two main structures that are used within that compiler: the high level data structure (HLDS) and the low level data structure (LLDS). These structures are handled, transformed, adapted by each of the compiler passes, and at first sight these structures seem ideal candidates for being updated destructively. Other structures, such as lists, maps, trees, also offer a great potential for structure reuse.

[7] Currently, the CTGC system is implemented in a separate branch of the compiler.

12.8 Conclusion

A major contribution of this work is the integration of the CTGC system in the Melbourne Mercury Compiler and its evaluation. Some small benchmarks were used, but also two medium-sized real-life and more complex programs were considered: a ray tracer (mostly deterministic code) and a finite domain solver (with non-deterministic search procedures).

For all our benchmarks we obtain considerable memory savings (up to 50% for the ray tracer) depending on the CTGC configuration used. We observe that in general the best results are obtained with the graph based selection strategy, while allowing reuses with differing arities between the deconstructions and the constructions is not recommended in the presence of cell caching. The CTGC system performs well, even in the presence of non-deterministic code, although of course the reuse within the non-deterministic code itself remains inherently limited.

The experiments make clear that time speedup is not necessarily a consequence of memory savings as this depends on the run-time system. A good cooperation with the run-time garbage collector becomes mandatory.

Chapter 13

Optimisation Derivation System

In the previous chapters we have taken as a heuristic that for each procedure at most two versions are generated: a version that is always safe to call (and that at most implements the *unconditional* forms of data structure reuse), and a version that implements all forms of detected reuses yet that imposes a number of conditions on calls to that version. This heuristic is simple and safe, yet may lead to poor results if *too many* opportunities for reuse in procedures are detected w.r.t. the actual uses of these procedures. In such cases we want to be able to find more refined versions instead of only these two.

In this chapter we develop a so called *optimisation derivation framework*. The idea is to derive descriptions of possible optimisations within a procedure in a goal-independent way. This characterisation of the optimisation opportunities gives a basis to a better understanding of the optimisation possibilities of the code such that better heuristics can be derived, and more versions can be generated, but having the guarantee that no two versions with same optimisations will ever be created.

The framework is developed in a generic way using again the denotational approach. We instantiate it for the domain of reuse information and discuss the results.

13.1 Introduction

A recurring problem that most optimising compilers or program transformers face is the problem of generating the adequate versions for each program entity that can be optimised in order to obtain a sufficiently optimised program (Mazur,

Ross, Janssens, and Bruynooghe 2001; Vanhoof and Bruynooghe 1999; Leuschel, Martens, and De Schreye 1998; Henglein and Mossin 1994; Puebla and Hermenegildo 1999a). In a logic programming setting, the entities to be optimised are for example predicates or in our case procedures. If each procedure can be optimised in a number of different ways, it is important to know which versions are interesting. If too many versions are generated, the size of the transformed program may become too large with respect to the efficiency gained by the optimisations that these versions allow. On the other hand, if the version generation policy is too restrictive, the program may be suboptimal, and still not as efficient as one might have hoped.

A classic way of guiding the version generation process during the transformation of a program is to use a top-down program analysis system that is able to detect the different uses of a predicate (Puebla and Hermenegildo 1999a; Vanhoof and Bruynooghe 1999; Leuschel, Martens, and De Schreye 1998). For each use of a predicate, an adequate optimised version is generated leading to the so called *multiple specialisation* of that predicate. Usually, this top-down approach still generates too many versions, therefore a separate pass dealing with this multiple specialisation of the program may be needed (Puebla and Hermenegildo 1999a; Winsborough 1992). Another disadvantage of these approaches is that for each new use of a predicate — characterised by a different call description, the program needs to be reanalysed as the underlying analyses can not guarantee that the optimisations are safe for call descriptions differing from the descriptions with which the predicate was already analysed. Finally, none of these global approaches work well in the presence of modules.

In our work on CTGC, a typical example of program optimisation, we use a different approach. For each procedure we generate at most two versions: a plain non-optimised version that is always safe to call, and a fully optimised version that can only be used if the caller of the predicate meets the (harsh) conditions in order to guarantee that all the optimisations are safe. Obviously, there is no risk for code explosion, yet a lot of intermediate opportunities for optimisations are missed, as was illustrated by some of our benchmarks.

The common problem in the above approaches is that, although we are able to spot the optimisations, we do not have an adequate mechanism for collecting and comparing the possible optimisations *before* the actual versions are created[1]. Therefore, we have developed an *optimisation derivation system* which is actually an analysis tool that is capable of spotting possible optimisations and relating these optimisations to the calls for which they are safe. Indeed, if we can collect the set of possible optimisations within a predicate and relate each optimisation to a requirement on the call descriptions of that predicate, then we can (automatically) reason about the versions that are interesting to generate. Some examples:

[1] In (Puebla and Hermenegildo 1999a) the optimisations are compared once all the versions have been characterised.

- If more than one optimisation is spotted for the same call description, then this may be a reason for generating a version with the corresponding optimisations. Translated in terms of CTGC, if two deconstructions (with matching constructions that can reuse the deconstructed data structures) are characterised by the same reuse information tuple, then a version implementing both reuses might be of interest, as two reuses are obtained with only one single reuse information tuple limiting the use of the obtained reuse version of the procedure.
- For recursive procedure definitions it is more interesting to enable optimisations that can also be performed in each of the recursive calls then to obtain versions in which only the first call of the procedure has interesting optimisations enabled, while none of the recursive calls in that call can profit from these optimisations. In terms of CTGC in the context of a list-manipulating procedure, it can for example be more interesting to focus on the reuses of the list-cells that are consecutively deconstructed in each of the recursive calls, then to envisage reuses of spurious small data structures.

Moreover, the relation between possible optimisations and a description of the calls that allow them is also interesting feedback to the programmer: Why can an atom at some program point not be optimised? How does a predicate need to be called in order to profit from the full optimisation potential? Finally, it can also be seen as a step towards conceptually separating the analysis of a predicate from the version generation process which depends on it, opening the field for better insights into the latter problem.

We assume that the input to our optimisation derivation system is a goal-independent annotation table containing annotations that are needed to decide the intended optimisations. In the context of CTGC, where intended memory optimisations depend on liveness analysis which itself depends on possible structure sharing and forward/backward use information, we therefore expect that all underlying information is already at hand before reasoning about the reuse information that depends on it. As already mentioned, we assume that these annotations are goal-independent and that they can be used to approximate goal-dependent situations using the appropriate combination operator, in this text consistently denoted by comb. This is not too much of a restriction, as we want to continue to be able to support modular programming which requires a goal-independent based approach to program analysis anyway. The precision of the resulting approximated goal-dependent descriptions has an effect on the precision of the optimisations that can be derived. For domains that satisfy the equivalence conditions required by Theorem 5.4 we have a guarantee that the goal-dependent information based on the goal-independent annotations will always be as precise as the goal-dependent information derived by a separate goal-dependent analysis. In the case of our structure sharing analysis, we have that

the underlying concrete structure sharing domain satisfies the equivalence conditions, hence approximating the information in a goal-dependent setting is as correct as approximating it in the goal-independent based setting. Moreover, as the abstract structure sharing domain is not idempotent, the results of the differential approach as well as the results of the goal-independent based approach will in general be more precise then the results obtained in a goal-dependent setting. Nevertheless, in the optimisation derivation framework derived in this chapter we do not explicitly require the basic domain to satisfy the equivalence conditions, although of course, it can be an advantage if it is.

Given a specification of the intended type of optimisation and a correctly annotated program, our analysis is able to identify all the literals within the program that can *potentially* be optimised with respect to this specification. This means that our optimisation derivation system intrinsically *overestimates* the optimisation opportunities. As a consequence extra care is needed when using the obtained results for generating the appropriate safe versions. In general, this may mean that some of the requirements for optimisation may have to be verified again. It is possible to avoid this overestimation, but then we need a way to limit the number of optimisations we keep, hence, we need heuristics limiting the number of versions, which ultimately leads us back to the original versioning problem.

The main contribution of this chapter is a framework that embodies our novel approach to the characterisation of the optimisation opportunities within a predicate. We identify each optimisable literal by a description of the calls for which the optimisations are safe. Therefore, given a specific call description it becomes straightforward to verify what optimisations it allows.

13.2 Concrete and Abstract Domains

We use the top-down collecting semantics (Cousot and Cousot 1977) for which the concrete semantics of logic programs is expressed in terms of elements from a domain $\mathcal{C}$. As usual, $\mathcal{C}$ is required to be a complete lattice: $\langle \mathcal{C}, \subseteq, \cup, \cap, \top_{\mathcal{C}}, \bot_{\mathcal{C}} \rangle$. The abstract counterpart of $\mathcal{C}$ is a domain $\langle \mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \top_{\mathcal{A}}, \bot_{\mathcal{A}} \rangle$. To ensure a terminating fixpoint computation, we require this domain to be Noetherian[2]. Both domains capture the information that is necessary for detecting the intended optimisations. While $\mathcal{C}$ captures the relevant parts of the program environments as they occur while executing the program, $\mathcal{A}$ approximates these environments at compile-time. In most cases this *relevant part* describes the variable bindings as they occur at each stage of the program, limited to the variables occurring in the context of the procedure that is being executed or considered. The domain

[2]Recall that Noetherian domains are domains in which all ascending chains have finite length (Marriott, Søndergaard, and Jones 1994), c.f. Chapter 2.

of idempotent variable substitutions but also the domain we mainly used in this thesis of existentially quantified ex-equations $\wp(Eqn^+)$ are typical concrete domains. As other elements of the run-time environment can be of interest too — such as structure sharing for our CTGC system, we do not limit ourselves to these two typical concrete domains.

We require that the concrete and abstract domains are connected by a monotone, strict (*i.e.*, $\gamma \bot_{\mathcal{A}} = \bot_{\mathcal{C}}$) and co-strict (*i.e.*, $\gamma \top_{\mathcal{A}} = \top_{\mathcal{C}}$) concretisation function $\gamma : \mathcal{A} \to \mathcal{C}$. Note that the strictness property was not required in our earlier chapters, yet here it will be used to characterise the absence of possibilities of optimisation. We use the same terminology of call descriptions and exit descriptions depending on whether the descriptions represent the calls of specific procedures or the exit situation after completion of a call to a specific procedure. We also use the same terminology and notation of safe approximation: let $\delta \in \mathcal{A}$, and $\sigma \in \mathcal{C}$, then δ is said to be a *safe approximation* of σ, denoted by $\delta \propto \sigma$, iff $\sigma \subseteq \gamma(\delta)$.

We assume that both the concrete and abstract domain are used in the context of a goal-independent based semantics of the Mercury language. In that context, let $A^{\mathcal{C}}$ denote the goal-independent annotation table obtained by instantiating $\mathcal{S}em_{M\star p}$ with the concrete domain $\mathcal{C}$ and auxiliary operations $\{\mathsf{init}^{\mathcal{C}}, \mathsf{comb}^{\mathcal{C}}, \mathsf{add}^{\mathcal{C}}\}$, and $A^{\mathcal{A}}$ denote the goal-independent annotation table for the abstract domain $\mathcal{A}$ in the semantics $\mathcal{S}em_{M\star p}$ with the instantiated operations $\{\mathsf{init}^{\mathcal{A}}, \mathsf{comb}^{\mathcal{A}}, \mathsf{add}^{\mathcal{A}}\}$. If $\sigma \in \mathcal{C}$, resp. $\delta \in \mathcal{A}$, represents an actual call description of a procedure p, and $i \in \mathsf{pp}(p)$, then $\mathsf{comb}^{\mathcal{C}}(\sigma, A^{\mathcal{C}}(i))$, resp. $\mathsf{comb}^{\mathcal{A}}(\delta, A^{\mathcal{A}})$, correctly represents the call description at program point (i) (*before* the literal at (i) is actually performed) when p is called with call description σ, resp. δ. Note that the particular selection rule is implicitly present through the goal-independent annotation tables.

We provide the following explicit definition of the safeness of the goal-independent annotation $A^{\mathcal{A}}$ w.r.t. $A^{\mathcal{C}}$.

Definition 13.1 (Safeness) *Let p be a procedure such that program point $i \in \mathsf{pp}(p)$, then the goal-independent annotation $A^{\mathcal{A}}(i)$ is safe w.r.t. the goal-independent annotation $A^{\mathcal{C}}(i)$ and the combination operations $\mathsf{comb}^{\mathcal{A}}$ and $\mathsf{comb}^{\mathcal{C}}$ for the abstract, resp. concrete domains, if and only if $\forall \sigma \in \mathcal{C}, \delta \in \mathcal{A}$ where $\delta \propto \sigma$, we have $\mathsf{comb}^{\mathcal{C}}(\sigma, A^{\mathcal{C}}(i)) \propto \mathsf{comb}^{\mathcal{A}}(\delta, A^{\mathcal{A}}(i))$.*

A goal-independent annotation table $A^{\mathcal{A}} : \mathsf{pp} \to \mathcal{A}$ is safe w.r.t. a concrete annotation table $A^{\mathcal{C}} : \mathsf{pp} \to \mathcal{C}$ and the combination operators $\mathsf{comb}^{\mathcal{A}}$ and $\mathsf{comb}^{\mathcal{C}}$ iff each goal-independent annotation within this annotation table is safe w.r.t. the concrete annotation.

13.3 Intuitive Example

Suppose we want to identify the unifications of the form $X = f(Y_1, \ldots, Y_n)$ that can be compiled into *deconstructions*, assuming that some interesting optimisation can be done for such unifications. Note that this is a pure hypothetical setting, and has no other purpose than to illustrate our approach as such information is normally available through the mode-information declared and inferred in Mercury programs. One possible way to discover and optimise such deconstructions is to use a standard polyvariant groundness analysis (Marriott and Søndergaard 1993; Heaton, Abo-Zaed, Codish, and King 2000; Howe and King 2000) system. But such analyses are typically used to derive success descriptions for predicates, while we are interested in deriving specifications of call descriptions.

Consider the code of predicate append in general, thus assuming general unifications instead of specialised ones:

```
append(X,Y,Z) :-
        (
                (1) X = [],
                (2) Z = Y
        ;
                (3) X = [Xe|Xs],
                (4) append(Xs,Y,Zs),
                (5) Z = [Xe|Zs]
        ).
```

Our goal is to describe the call descriptions for append for which some or all of the unifications, either in the first call or in its recursive calls, are deconstruction unifications.

In order to decide when unifications become deconstructions, mode information is needed, *i.e.*, we need to be able to determine when variables become ground. In the concrete domain, this information can be expressed using existential term-equations which we can abstract by either using the domain of definite boolean functions *Def* (Howe and King 2000), or the more precise domain *Pos* (c.f. Chapter 5). Let $Def_\bot$ and $Pos_\bot$ be the domains *Def* and *Pos* resp., extended with the bottom element false, then these domains are complete lattices: $\langle Def_\bot, \models, \vee, \wedge, \mathsf{true}, \mathsf{false}\rangle$ and $\langle Pos_\bot, \models, \vee, \wedge, \mathsf{true}, \mathsf{false}\rangle$. Both domains can be related to the concrete domain $\wp(Eqn^+)$ using the concretisation function defined for $Pos_\bot$ in Example 5.7 (Page 73). Recall that for both domains false represents the empty set of concrete variable bindings, hence reflects a failing derivation.

Figure 13.1 gives the goal-independent annotation table for append in terms of the abstract domain $Def_\bot$ but also in terms of the domain $Pos_\bot$ as in the latter the same table would have been obtained. We denote this table using A^g, where g is used as a superscript to refer to the groundness information that it records. Both domains use the same combination operation, namely $\mathsf{comb}^{Pos_\bot} = \mathsf{comb}^{Def_\bot} = \wedge$

(c.f. Example 5.7). This table is safe w.r.t. $\wp(Eqn^+)$ and the concrete combination operation $\mathsf{comb}^{\wp(Eqn^+)}$ (Definition 5.6). Indeed, if $\delta \in Pos_\perp$ is a call description of the procedure append, and $i \in \mathtt{pp}(\mathtt{append})$, then $A^g(i) \wedge \delta$ is a safe description of the groundness information for each individual program point (i).

pp	$A^g(\mathtt{pp})$
1	true
2	x
3	true
4	$x \leftrightarrow x_e \wedge x_s$
5	$(x \leftrightarrow x_e \wedge x_s) \wedge (z_s \leftrightarrow x_s \wedge y)$

Figure 13.1: Goal-independent annotation table A^g for predicate append in terms of $Pos_\perp$ (and $Def_\perp$).

Using the goal-independent annotation table, we want to be able to relate call descriptions with the intended optimisations. As these optimisations are only possible if a unification turns out to be a deconstruction, we need to express this fact as a kind of condition that needs to be satisfied by the literal. In our case, a unification $X = f(Y_1, \ldots, Y_n)$ is a deconstruction if X is ground. In terms of the domains $Pos_\perp$ and $Def_\perp$ this means that if the abstract call description $\delta_i \in Pos_\perp$ (or $Def_\perp$) where $i = \mathtt{pp}(X = f(Y_1, \ldots, Y_n))$ is such that $\delta_i \models x$, then the unification is indeed a deconstruction. The condition of X needing to be ground, here expressed by the simple abstract description $x \in Pos_\perp$ (or $Def_\perp$), is called the *minimal requirement* for optimising the unification $X = f(Y_1, \ldots, Y_n)$. We shall systematically denote descriptions used in the role of minimal requirement by the letter μ. The columns in Figure 13.2 and Figure 13.3 labelled $\mu_{\mathtt{pp}}$ (under "iter 1") show the (renamed) minimal requirements for the unifications in append to become deconstructions.

Given these minimal requirements expressed at the level of the individual unifications, we now need to find how to reason about the call descriptions for append that would allow the optimisations implied by these minimal requirements. Consider a call description δ for append. We will now study the unification at program point (3). From the safety of the goal-independent annotation table, we can check whether the unification at (3) is a deconstruction by verifying

$$A^g(i) \wedge \delta \models \mu_i \tag{13.1}$$

for $i = 3$. We can use Equation (13.1) for deriving the *most general* call description δ'_i for which μ_i is satisfied and thus for which the optimisation is safe. Such a *most general* call description is interesting as indeed, for each call description δ that is subsumed by that most general description, optimisation is automatically allowed: $\forall \delta \models \delta'_i$, if δ'_i satisfies Equation (13.1), then so does δ.

	iter1		iter2	
pp	$\mu_{\mathtt{pp}}$	$\delta^m_{\mathtt{pp}}$	$\mu_{\mathtt{pp}}$	$\delta^m_{\mathtt{pp}}$
1	x	x	x	x
2	$-$	$-$	$-$	$-$
3	x	x	x	x
4	$-$	$-$	true	true
5	z	z	z	z
		$x \vee z = \mathsf{true}$		true

Figure 13.2: Gathered $Def_{\perp}$-information for predicate append. $\mu_{\mathtt{pp}}$ are minimal requirements, $\delta^m_{\mathtt{pp}}$ are call requirements. In $Def_{\perp}$ we obtain that every call has a possibility of optimisation, hence an overestimation.

	iter1		iter2	
pp	$\mu_{\mathtt{pp}}$	$\delta^m_{\mathtt{pp}}$	$\mu_{\mathtt{pp}}$	$\delta^m_{\mathtt{pp}}$
1	x	x	x	x
2	$-$	$-$	$-$	$-$
3	x	x	x	x
4	$-$	$-$	$x_s \vee z_s$	x
5	z	z	z	z
		$x \vee z$		$x \vee z$

Figure 13.3: Gathered $Pos_{\perp}$-information for predicate append. $\mu_{\mathtt{pp}}$ are minimal requirements, $\delta^m_{\mathtt{pp}}$ are call requirements. In this domain, append can be optimised if either X or Z is ground.

The point is now to define that *most general* call description w.r.t. a given other description. In the literature (Giacobazzi and Scozzari 1998; King and Lu 2002) the *most general* call description w.r.t. the bottom element of the lattice domain is usually called the *pseudo-complement* of that bottom element. Finding the most general call description w.r.t. any other element of the lattice is called the *relative pseudo-complement*. In both cases, the pseudo-complement is specifically defined in terms of the greatest lower bound operation defined in that lattice. As we are interested in finding most general descriptions w.r.t. any other combination operation defined in the lattice domain, we generalise the notion and define a *general relative pseudo-complement* of the specific call description w.r.t. a specific combination operation, which we will simply abbreviate to *general pseudo-complement*.

In the context of the domain $Pos_{\perp}$, the combination operation with which call descriptions are combined is simply the greatest lower bound operation, hence the pseudo-complement operation is identical to the general pseudo-complement, and is in this case simply the logical implication (Giacobazzi and Scozzari 1998).

Therefore, if δ'_i is meant to represent the general pseudo-complement of a local description given by a goal-independent annotation table A^g at program point (i), then δ'_i can be computed as $A^g(i) \rightarrow \mu_i$, where μ_i is the minimal requirement at that program point. Intuitively, in our running example, for $i = 3$, the situation is rather trivial in the sense that it suffices for X to be ground to obtain a deconstruction. As X is a head variable, this simply translates itself to the description x for calls to append. This is confirmed by the (generalised) pseudo-complement. We have $A^g(3) = \mathsf{true}$, $\mu_3 = x$, and therefore $\delta'_3 = \mathsf{true} \rightarrow x = x$.

In general, pseudo-complements may involve any variable of the procedure. Take for example the following procedure definition:

```
%:- pred second(list(int),list(int)).
%:- mode second(in,out) is semidet.
second(L1,L):-
        (1) L1 = [X | L2],
        (2) L2 = [Y | L3],
        (3) Z is X + Y,
        (4) L3 = [ Z | L3 ].
```

Let us consider the second unification. This unification becomes a deconstruction when $L2$ is ground. Hence, we have the minimal requirement: l_2. The goal-independent annotation at that unification consists of the description $A^g(2) = l_1 \leftrightarrow (x \wedge l_2)$. The generalised pseudo-complement of $A^g(2)$ w.r.t. the minimal requirement l_2 is the expression $l_1 \leftrightarrow (x \wedge l_2) \rightarrow l_2$, which is equivalent to $(l_1 \vee l_2)$. Indeed, it suffices that l_1 or l_2 is ground in order for the second unification to be a deconstruction. The variables of that generalised pseudo-complement are $\{l_1, l_2\}$, hence also local variables can appear in such expressions. Yet, as these pseudo-complements are used in fixpoint computations, we need to have only head variables appearing in our resulting most general call descriptions. This can be achieved by projecting the local variables away from the obtained expression. For this purpose we use the so called *universal projection* with which we obtain a description that subsumes the original description. In this particular example we have: $(l_1 \vee l_2)|_{\{l_1,l\}} = \forall_{\overline{\{l_1,l\}}} l_1 \vee l_2 = \forall_{\{l_2\}} l_1 \vee l_2 = l_1$ and indeed, $l_1 \models l_1 \vee l_2$. We call the resulting universally projected pseudo-complements *call requirements*, as they describe a requirement in terms of the variables of the head of the procedure such that if that requirement is fulfilled, optimisation of the literal within the procedure is safe.

Figure 13.2 and Figure 13.3 list the call requirements for each of the unifications of interest (column "iter 1", δ^m_{pp}) in the append-procedure. The call requirements can have two extreme values: true — the optimisation is always possible as all concrete call descriptions belong to the concretisation set of the abstract value true, and false — no call description can ever allow the safe optimisation of the involved literal as the concretisation of false is the empty set of concrete ex-equations representing the concrete situation of failure. Note that in $Def_\perp$, not

all abstract descriptions have a pseudo-complement, so in some cases approximation may be needed.

The above presented intuitions are mainly for literals with explicit possibilities of optimisations. Of course, if a procedure has some potential optimisation at one of its literals, then calls to that procedure may have that same potential of optimisation, hence minimal requirements and therefore call requirements should be derived too. We present the intuition using our example of append.

From Figure 13.2 and Figure 13.3 we deduce that some of the unifications within a call to append(X,Y,Z) become deconstructions if either X is ground, or Z. This is represented in the columns labelled "iter 1", δ^m_{pp}. We can use this information to check the optimisation possibilities of the recursive call in append.

In general, each procedure may have several opportunities for optimisation, each opportunity characterised by its own call requirement. As a recursive call requires a fixpoint computation, we must ensure finiteness of the computation, hence we do not propagate each call requirement individually, but use the least upper bound of the call requirements instead. The result is one single call requirement that represents the calls for which some optimisation *may* be allowed. Hence, we obtain a weak form of optimisation information. Another possibility of reducing the call requirements into one single description is by using the greatest lower bound. This operation guarantees that if a call description is subsumed by the resulting call requirement, then *all* detected optimisations are safe. Yet, this approach is too restrictive as this allows only calls to optimised versions of the procedures if *all* optimisations can be performed. This results in the same approach as we used in our CTGC system, namely to compile each procedure into at most two versions: a version without restrictions in use, and a version with full optimisation yet with many restrictions imposed on its use. As this behaviour is exactly the behaviour we want to avoid, we choose the liberal approach, and decide to propagate optimisation requirements by their least upper bound. A viable less restricting alternative is presented in Section 13.5.

For append the call requirements for optimising the unifications are x and z. The resulting least upper bound is $x \vee z$ in $Pos_\perp$ — if either X or Z is ground optimisation is possible, and true in $Def_\perp$ — any call may have some potential of optimisation. While $x \vee z$ is a good and perfect estimation, true is a clear overestimation. This illustrates the influence of the choice of the abstract domain on the obtained results as well as the fact that using the least upper bound as our combination operator for individual call requirements inherently *weakens* the descriptions of the possible optimisations. Using this resulting minimal requirement for the recursive call, we again use Equation (13.1) to compute a new call requirement: in $Pos_\perp$, at program point (4), the pseudo-complement of $A^g(4) = (x \leftrightarrow x_e \wedge x_s)$ w.r.t. $\mu_4 = x_s \vee z_s$ (notice the renaming) is $\delta'_4 = x \vee x_s \vee z_s$, which, after universal projection on the head variables, yields $\delta^m_4 = \delta'_4|_{\{x,y,z\}} = x$. Note that it is not surprising that the result is not $x \wedge z$ as taking the left-to-right selection rule, and

thus interpreting the code in the order in which it is written, whether append is called with Z ground or not, the recursive call will always have a free variable as its third argument[3].

The result of collecting the optimisations for append can be read from the columns labelled δ^m_{pp} ("iter 2") and is as follows: Call descriptions in which X is ground allow the optimisation of the literals at program points (1), (3) and (4); call descriptions in which Z is ground allow only the optimisation of the unification at program point (5). In $Def_{\perp}$, by overestimation, we also obtain that the recursive call of append (program point (4)) might be optimisable for *any* call description, regardless of the groundness in that description.

We can use these results in a number of different ways:

- Using the results obtained in $Pos_{\perp}$, we may be able to (automatically) see that when the first argument is ground, more optimisations can be performed. Hence, it may be interesting to generate a version for that particular case.
- When generating that specialised version, we only need to keep the optimisation requirement x, and compare all actual calls to append with that requirement.
- When generating that specialised version, and thus actually optimise the involved literals, we may need to verify each of the requirements at these literals for calls satisfying the requirement x. This is as a form of extra control to guarantee the safeness of the resulting version.
- Obviously, the results obtained in $Def_{\perp}$ are too coarse for deriving actual interesting information.

The formalisation and generalisation of these ideas are presented in the following section.

13.4 Optimisation Derivation System

13.4.1 Basic Components

The optimisation derivation system starts with pre-defining clear conditions that formalise when some literals can be optimised. We call the literals for which such conditions are pre-defined the *base atoms*. The other literals are called *non base atoms*. If a non base atom is a built-in, then it is called a *non base built-in atom*. If a non base atom is a call to a user defined procedure, it is called a *non base call atom*. In essence, we use the following subdivision for the set of literals:

$$Literal = BaseAtom \cup NBBuiltin \cup NBCall$$

[3]Recall that this is a hypothetical setting.

where *BaseAtom*, *NBBuiltin* and *NBCall* denote the set of base atoms, the set of non base built-in atoms and the set of non base call atoms respectively. This subdivision is important because optimisation information is *generated* at base atoms and *propagated* through non base call atoms. Non base built-in atoms are neutral w.r.t. the intended optimisations. For simplicity, we use the notation $p(X_1, \ldots, X_n)$, where $\{X_1, \ldots, X_n\} \subseteq \mathcal{V}$ and $p \in \Pi$, to refer to literals in general, therefore not distinguishing unifications from other literals.

Example 13.1 *In our groundness example, all unification literals are base atoms, and all procedure calls are non base call atoms. In our CTGC system, the purpose is to optimise deconstructions by reusing the memory associated with the involved data structure for other data structures. Hence, only deconstruction unifications are considered as base atoms. All other unifications are non base built-in atoms, while all remaining literals are non base call atoms.*

For each program we assume that a goal-independent annotation table is available. This annotation table is expressed in terms of an abstract Noetherian domain $\langle \mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \bot_{\mathcal{A}}, \top_{\mathcal{A}} \rangle$. The annotations describe the relevant goal-independent information enabling the characterisation of the subsequent optimisations. We assume that the annotations are safe w.r.t. a concrete domain $\langle \mathcal{C}, \subseteq, \cup, \cap, \bot_{\mathcal{C}}, \top_{\mathcal{C}} \rangle$, the abstract combination operation $\mathsf{comb}^{\mathcal{A}}$ and concrete combination operation $\mathsf{comb}^{\mathcal{C}}$. We require that the abstract domain $\mathcal{A}$ must be suitable to express the conditions of optimisation of the literals of interest, *i.e.*, must be expressive enough to formulate the so called *minimal requirements* for optimising the base atoms in the language. Elements in $\mathcal{A}$ are related to elements in $\mathcal{C}$ through a so called concretisation function γ. The tuple $(\mathcal{C}, \gamma, \mathcal{A})$ is required to be an insertion (Definition 5.1). As an extra we also require γ to be strict (instead of co-strict only).

Definition 13.2 (Requirement, Minimal Requirement) *A* requirement *for the optimisation of a literal $p(X_1, \ldots, X_n)$ is an abstract description $\mu' \in \mathcal{A}$, such that for all concrete descriptions in $\gamma(\mu')$, the optimisation of the literal is allowed. Formally: $\forall \sigma \in \mathcal{C}$, if $\sigma \subseteq \gamma(\mu')$ then $p(X_1, \ldots, X_n)$ can be optimised. A requirement μ is called* minimal *if it is the* largest *requirement for optimisation of that literal, formally, $\forall \mu' \in \mathcal{A}$, if μ' expresses a requirement for optimisation, and $\mu' \sqsubseteq \mu$, then μ is a minimal requirement.*

We use the terminology of *minimal* in the sense that it states that descriptions must at least satisfy the conditions expressed by μ in order to obtain a safe characterisation of the optimisation involved.

Lemma 13.1 *If μ is a minimal requirement for a literal, then the literal can be optimised $\forall \delta \in \mathcal{A}$ for which $\delta \sqsubseteq \mu$.*

This is a consequence of the monotonicity of γ. In the following sections, we mainly use this formulation.

The minimal requirements for base atoms need to be predefined either by hand when instantiating the optimisation derivation framework, or by some automatic system preceding the optimisation derivation phase. We assume that the minimal requirements for base atoms are recorded and stored in a so called *base table*.

Definition 13.3 (Base Table) *A* base table*, usually denoted by the letter B, is a function mapping base atoms modulo variance (which we denote by subscripting BaseAtom with the symbol $\approx$) to their minimal requirements*

$$\mathit{BaseTable} : \mathit{BaseAtom}_{\approx} \rightarrow \mathcal{A}$$

The goal of the optimisation derivation system is to derive minimal requirements for non base call atoms. Minimal requirements are explicitly defined for literals and express the possibility of optimising the literal it belongs to. In the process of propagating the optimisation opportunities we also need to reason about the optimisations at the level of a procedure. For this purpose we introduce the notion of *call requirement* which expresses the possibility of optimising a literal within the procedure definition in terms of the head variables of that procedure.

Definition 13.4 (Call Requirement, Minimal Call Requirement) *A* call requirement *for a procedure p with respect to a program point $i \in \mathsf{pp}(p)$ is an abstract description, systematically denoted by δ_i^m, such that for each concrete call to p subsumed by δ_i^m, the optimisation of the literal at program point (i) in the definition of p is allowed. Formally: $\forall \sigma \in \mathcal{C} : \sigma \subseteq \gamma(\delta_i^m)$, literal l_i can safely be optimised. A call requirement is* minimal *if there does not exist a larger call requirement w.r.t. the order relation $\sqsubseteq$ in $\mathcal{A}$.*

In the following we use *call requirement* as a synonym for *minimal call requirement*.

Finally, the abstract domain is required to have a *generalised relative pseudo-complement* operator which is defined with respect to the combination operators used for the goal-independent annotations.

Definition 13.5 (Generalised Pseudo-Complement) *The* generalised pseudo-complement *of an abstract description δ_a relative to an abstract description δ_b and w.r.t. the combination operator $\mathsf{comb}^{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{A} \rightarrow \mathcal{A}$, denoted as $\delta_a \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} \delta_b$, is a description $\delta_c' \in \mathcal{A}$ such that:*

$$\begin{aligned} & \delta_c' = \bigsqcup \{ \delta_c \in \mathcal{A} \mid \mathsf{comb}^{\mathcal{A}}(\delta_a, \delta_c) \sqsubseteq \delta_b \} \\ \textit{and} \quad & \mathsf{comb}^{\mathcal{A}}(\delta_a, \delta_c') \sqsubseteq \delta_b \end{aligned}$$

This is a generalisation of the definition of a *pseudo-complement* as it is defined in terms of a general combination operator $\mathsf{comb}^{\mathcal{A}}$ instead of the greatest lower bound operation $\sqcap$ of the abstract domain (Giacobazzi and Scozzari 1998).

The pseudo-complement has an interesting property for concrete domains having a bottom element representing a failing derivation. Indeed, if the concrete domain has such a bottom element, then by the strictness of the concretisation function (*i.e.*, $\gamma(\bot_{\mathcal{A}}) = \bot_{\mathcal{C}}$), we know that if the generalised pseudo-complement is $\bot_{\mathcal{A}}$, then the optimisation is *never* possible as there is no non-failing concrete description that is correctly described by $\bot_{\mathcal{A}}$. For such situations, it is useful to require that $\bot_{\mathcal{C}}$ does represent failure. This is not a restriction in itself. If the domain did not contain such a bottom element, then it can always be added. Having such an element ensures that the set $\{c \in \mathcal{A} \mid \mathsf{comb}^{\mathcal{A}}(a, c) \sqsubseteq b\}$ in Definition 13.5 is never empty as it will always at least contain $\bot_{\mathcal{A}}$. This makes the definition of the pseudo-complement and its use more compact as there is no further discussion needed on whether this set is empty or not, *i.e.*, whether there exists at least one call description δ_c such that $\mathsf{comb}^{\mathcal{A}}(\delta_a, \delta_c) \sqsubseteq \delta_b$ is satisfied or not. Whether or not δ'_c satisfies $\mathsf{comb}^{\mathcal{A}}(\delta_a, \delta'_c) \sqsubseteq \delta_b$ depends on the abstract domain, and in some cases approximation may be needed. This was already illustrated for the particular domain $Def_{\bot}$.

13.4.2 Basic Framework

In the process of deriving optimisations we compute call requirements from minimal requirements. We do this by using the generalised relative pseudo-complements of the goal-independent annotations of a literal with respect to the minimal requirement defined or computed for that literal. We formalise the actual optimisation derivation process using denotational semantics (Nielson and Nielson 1992; Allison 1986) with which we relate the optimisation opportunities to the syntactic objects constituting a program. Optimisations are *generated* at base atoms, and *propagated* at non base call atoms, independent of an exact call description. This behaviour is similar to the behaviour described by the goal-independent part of the goal-independent based semantics defined for Mercury in Section 5.7 with the difference that instead of differentiating between unifications and call atoms, here we make a distinction between base atoms and non base call atoms. Therefore, we describe our optimisation derivation framework as a variation on the $\mathcal{S}em_{M\star}$ semantics.

As we argued above, the abstract domain used to express the requirements is the same abstract domain as the one used for the goal-independent annotations on which the optimisation derivation system depends. Let $\langle \mathcal{A}, \subseteq, \cup, \cap, \bot_{\mathcal{A}}, \top_{\mathcal{A}} \rangle$ be that abstract domain for which a suitable combination operation $\mathsf{comb}^{\mathcal{A}}$ is defined and a generalised pseudo-complement w.r.t. that combination operation is given. As the optimisation derivation is based on a goal-independent annota-

tion table, say A, expressing the underlying goal-independent properties of each of the literals in the program, and a base table B expressing the minimal requirements for the base atoms in the program, we could thread these tables explicitly along each of the semantic functions in $\mathcal{S}em_{M\star}$. Yet to not to clutter the notation, we assume that these tables are implicitly present and can be queried using the following functions:

$$\begin{array}{lcl} \mathsf{gi} & : & \mathsf{pp} \to \mathcal{A} \\ \mathsf{gi}(i) & = & A(i) \end{array} \tag{13.2}$$

and

$$\begin{array}{lcl} \mathsf{base} & : & \mathit{BaseAtom} \to \mathcal{A} \\ \mathsf{base}(p(\overline{X})) & = & \text{let } (p(\overline{Y}), \delta) \in B \text{ in} \\ & & \quad \rho_{\overline{Y}\to\overline{X}}(\delta) \end{array} \tag{13.3}$$

We assume that base takes care of correctly renaming the minimal requirement of the base atom that is looked up.

We now define the *modified goal-independent semantics*, $\mathcal{S}em_{\omega}$, as the semantics consisting of the function clauses used in the definition of $\mathcal{S}em_{M\star}$, see Figure 5.8 (page 92) and Figure 5.9 (page 92), where all $_{M\star}$ subscripts are replaced by $_{\omega}$, yet where the clauses defining the semantics of literals are replaced by the following three clauses (here using δ to denote the optimisation requirements instead of the usual $\mathcal{S}$ to differentiate the requirements from normal descriptions). The semantics for literals is shown in Figure 13.4.

$$\begin{array}{lcl} \mathbf{L}_{\omega}[\![p(\overline{X})]\!]\, e\, \delta & = & \text{let } \delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\ & & \text{let } \mu_i = \mathsf{base}(p(\overline{X})) \text{ in} \\ & & \text{let } \delta_i = \delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\to} \mu_i \text{ in} \\ & & \quad \delta \sqcup \delta_i \\ & & \qquad \text{where } p(\overline{X}) \in \mathit{BaseAtom} \\ \mathbf{L}_{\omega}[\![p(\overline{X})]\!]\, e\, \delta & = & \text{let } \delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\ & & \text{let } \mu_i = e(p(\overline{X})) \text{ in} \\ & & \text{let } \delta_i = \delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\to} \mu_i \text{ in} \\ & & \quad \delta \sqcup \delta_i \\ & & \qquad \text{where } p(\overline{X}) \in \mathit{NBCall} \\ \mathbf{L}_{\omega}[\![p(\overline{X})]\!]\, e\, \delta & = & \delta \\ & & \qquad \text{otherwise} \end{array}$$

Figure 13.4: Semantics of literals in $\mathcal{S}em_{\omega}$.

Some remarks to this definition:

- In this definition, the auxiliary functions add and comb are absent. The add operation is indeed completely removed, yet the comb operation of the ab-

stract domain is still present through the generalised pseudo-complement of the domain. This means that the set of auxiliary functions used in $\mathcal{S}em_\omega$ only consists of the init function and the generalised pseudo-complement w.r.t. a combination operation.

- The formalisation for base atoms consists of consulting the base table to look up the minimal requirement for the base atom, computing the generalised pseudo-complement of the goal-independent annotation w.r.t. that minimal requirement, and combine the result with the already available call requirement.
- Similarly, non base call atoms are handled by looking up the minimal requirement for optimising the considered literal, computing the generalised pseudo-complement w.r.t. the goal-independent annotation of that literal, and combining the result with the already available call requirement.
- Call requirements are combined using the least upper bound operation $\sqcup$ instead of the greatest lower bound as we did in our strategy of producing only two versions of each of the encountered procedures in the CTGC system used in Chapter 12.
- If a literal can not be optimised then either this is due to the minimal requirement being $\bot_{\mathcal{A}}$ or the call requirement computed using the generalised pseudo-complement operation is $\bot_{\mathcal{A}}$. Both situations express the fact that no abstract description can be found for which all the concrete descriptions allow the intended optimisation. This behaviour is implicitly present in the given semantic rules.
- In the preceding intuitive example we detailed how call requirements are projected onto the head variables. In this definition this projection is still present yet instead of projecting the computed call requirements immediately we only do the projection at the level of the procedure semantics ($\mathbf{Pr}_\omega$) which includes the projection to the head variables of the obtained description. Note that this projection operation must be such that the resulting call requirement subsumes the original one.

Example 13.2 *The groundness propagation system described intuitively in the previous section can be formalised as an instantiation of $\mathcal{S}em_\omega$ with either the domain $\langle Pos_\bot, \models, \vee, \wedge, \mathsf{false}, \mathsf{true}\rangle$ or the domain $\langle Def_\bot, \models, \vee, \wedge, \mathsf{false}, \mathsf{true}\rangle$ as basic abstract domain. The auxiliary operations are then*

$$\begin{array}{lcllcl}
\mathsf{init}^{Pos_\bot} & = & \mathsf{init}^{Def_\bot} & = & \mathsf{false} \\
\mathsf{comb}^{Pos_\bot}(\delta_1,\delta_2) & = & \mathsf{comb}^{Def_\bot}(\delta_1,\delta_2) & = & \delta_1 \wedge \delta_2 \\
\delta_1 \overset{\mathsf{comb}^{Pos_\bot}}{\rightarrow} \delta_2 & = & \delta_1 \overset{\mathsf{comb}^{Def_\bot}}{\rightarrow} \delta_2 & = & \delta_1 \rightarrow \delta_2
\end{array}$$

In this formalisation, call requirements are immediately combined with each other, hence, for each procedure, one single call requirement is obtained. Due to the use of the least upper bound as the combination operator for call requirements, the resulting call requirement of a procedure only gives a *weak* description of the optimisation opportunities in its definition: if a call description meets the call requirement of a procedure, then this does not necessarily mean that any of its literals can be optimised. This is a drawback in the usefulness of the optimisation derivation framework as it demands a verification step to ensure the safeness of the final optimisations.

13.4.3 Variation

A variation on the previous basic framework is to keep the individual call requirements for a procedure separately, and only combine them when they are actually used for the minimal requirement of a literal calling that procedure. Although at the end the call requirements are also combined, at least we obtain the full list of call requirements for each procedure. The advantage is that the obtained list can be used for other purposes: verifying the most interesting optimisation criteria, verifying the harshest optimisation requirement, etc. This approach comes down to instantiating the above domain using the powerset of the initial abstract domain instead of the abstract domain itself. The ordering of this powerset follows naturally from the ordering of the underlying domain. If $\langle \mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \bot_{\mathcal{A}}, \top_{\mathcal{A}} \rangle$ forms the basic abstract domain, and $\langle \wp(\mathcal{A}), \subseteq, \cup, \cap, \{\,\}, \mathcal{A} \rangle$ the lattice of its powerset, and using Δ to denote sets of call requirements, then the clauses of the semantic functions for literals become as shown in Figure 13.5.

In this definition, the set of call requirements is only reduced when needed, *i.e.*, when computing the minimal requirement of a non base call atom. In this setting, the init-function should also represent a set of initial requirements.

13.4.4 Notions of Correctness

Although the final obtained results with $\mathcal{S}em_{\omega+}$ are as weak as in the previous formulation, now at least the intermediate separate call requirements are recorded and can thus be studied individually. If needed, also the program points can be recorded so as to be able to relate each call requirement with its literal from where it stems. In that case, we obtain a kind of program point annotation table recording the contribution of the corresponding literal to the optimisation possibilities of the procedure to which it belongs. We give a clear definition of such tables, provide an intended meaning for such tables obtained as a result of interpreting a Mercury program under the $\mathcal{S}em_{\omega+}$ semantics, and prove that the results with these semantics indeed respect this intended meaning.

$$
\begin{array}{lcl}
\mathbf{L}_{\omega+}[\![p(\overline{X})]\!]\, e\, \Delta & = & \text{let } \delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\
 & & \text{let } \mu_i = \mathsf{base}(p(\overline{X})) \text{ in} \\
 & & \text{let } \delta_i = \delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} \mu_i \text{ in} \\
 & & \quad \Delta \cup \{\delta_i\} \\
 & & \qquad \text{where } p(\overline{X}) \in \mathit{BaseAtom} \\
\mathbf{L}_{\omega+}[\![p(\overline{X})]\!]\, e\, \Delta & = & \text{let } \Delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\
 & & \text{let } \delta_{gi} = \sqcup \Delta_{gi} \text{ in} \\
 & & \text{let } \mu_i = e(p(\overline{X})) \text{ in} \\
 & & \text{let } \delta_i = \delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} \mu_i \text{ in} \\
 & & \quad \Delta \cup \{\delta_i\} \\
 & & \qquad \text{where } p(\overline{X}) \in \mathit{NBCall} \\
\mathbf{L}_{\omega+}[\![p(\overline{X})]\!]\, e\, \Delta & = & \Delta \\
 & & \qquad \text{otherwise}
\end{array}
$$

Figure 13.5: Semantic rules for literals in $\mathcal{S}em_{\omega+}$.

The function mapping program points of a procedure to call requirements allowing the optimisation of the associated literals represents the *local optimisation table* for that procedure. The table mapping the procedures of a rulebase onto local optimisation tables describing the possible optimisations within these procedures is called the *global optimisation table* for that rulebase.

Formally we have:

Definition 13.6 (Local and Global Optimisation Table) *The* local optimisation table *for a procedure p, denoted by ω_p is a mapping between some of the program points in p and call requirements describing the optimisations of the literals at these program points.*

The global optimisation table *of the rulebase r of a program, denoted by Ω_r, is a mapping between the procedures defined in r and the local optimisation tables derived for these procedures.*

The intended meaning of the global and local optimisation tables is defined as follows:

Definition 13.7 (Intended meaning of Ω_r) *Let Ω_r be the result of interpreting a rulebase of a program consisting of the procedures p_1 to p_{n_p} in $\mathcal{S}em_{\omega+}$ based on propagating sets of call requirements and using the rules in Figure (13.5). Let $(p_j, \omega_j) \in \Omega_r$ and $(i, \delta_i^m) \in \omega_j$. If a call description δ of p_j is such that $\delta \sqsubseteq \delta_i^m$, then, depending on the nature of the literal at program point i,* i.e., *l_i, we interpret this as:*

- *if $l_i \in \mathit{BaseAtom}$, then the literal can* definitely *be optimised for calls with call description δ.*

- *otherwise,* i.e., $l_i \in NBCall$*, the procedure corresponding to the called atom* might *allow some optimisations within it.*

For the semantics $\mathcal{S}em_{\omega+}$ to be acceptable we need to prove that it is *well defined*, *i.e.*, it can be computed by a terminating fixpoint computation, and *correct* w.r.t. the intended meaning of the derived optimisation table.

We first prove the following lemma which shows that the pseudo-complement operation is monotone, in particular in its second argument.

Lemma 13.2 *The generalised pseudo-complement* $\overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow}$ *is monotone in its second argument.*

Proof Let $c_1 = a \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} b_1$, $c_2 = a \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} b_2$, and $b_1 \sqsubseteq b_2$. We need to prove that $c_1 \sqsubseteq c_2$. By the definition of $\overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow}$ we have $\mathsf{comb}^{\mathcal{A}}(a, c_1) \sqsubseteq b_1$, and therefore by transitivity $\mathsf{comb}^{\mathcal{A}}(a, c_1) \sqsubseteq b_2$. This means that $c_1 \in C_2 = \{c \mid \mathsf{comb}^{\mathcal{A}}(a, c) \sqsubseteq b_2\}$. By definition we have $c_2 = \bigsqcup C_2$, and therefore $c_1 \sqsubseteq c_2$.

□

We now formulate the correctness of $\mathcal{S}em_{\omega}$ (and $\mathcal{S}em_{\omega+}$) w.r.t. the intended meaning of the resulting optimisation table (assuming sets of call requirements are propagated).

Theorem 13.1 *$\mathcal{S}em_{\omega+}$ is well defined w.r.t. the intended meaning of the global optimisation table (Definition 13.7) it defines for a rulebase r for a particular instantiation, if the instantiations of the auxiliary operations are monotonic.*

We only give an outline of the proof.

We assumed that our underlying abstract domain $\mathcal{A}$ is Noetherian (Page 298), hence it suffices to show that all auxiliary functions used in $\mathcal{S}em_{\omega}$ are monotone in order to guarantee well definedness (Nielson and Nielson 1992). Knowing that $\mathsf{init}^{\mathcal{A}}$ and $\mathsf{comb}^{\mathcal{A}}$ are required to be monotone anyway, and that the pseudo-complement is also monotone (Lemma 13.2), this is indeed the case.

The correctness of the semantics with respect to the definition of the intended meaning of a logic program in this setting follows from the correctness of the base table, the safeness of the goal-independent annotation table, and the definition of $\mathbf{L}_{\omega+}$ for base atoms and non base call atoms. The latter depends on the definition of the generalised pseudo-complement. This definition guarantees that if the requirement μ is minimal, then the call requirement δ^m computed from it will be minimal too. While for base atoms we use the exact minimal requirements as tabled in the base table, the minimal requirements used for non base call atoms might be approximations due to the use of the $\sqcup$ in the definition of $\mathbf{L}_{\omega+}$.

13.5 Increased Precision

While the above variation on the basic framework does not increase the overall precision hence strength of the results, here we propose a way to do so.

The precision can be strengthened if each call requirement is propagated individually. In that case, two call requirements would never be joined together. The definition of the literals would then be:

$$
\begin{array}{lcl}
\mathbf{L}_\Omega[\![p(\overline{X})]\!]\, e\, \Delta & = & \text{let } \delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\
 & & \text{let } \mu_i = \mathsf{base}(p(\overline{X})) \text{ in} \\
 & & \text{let } \delta_i = \delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} \mu_i \text{ in} \\
 & & \quad \Delta \cup \{\delta_i\} \\
 & & \qquad\qquad \text{where } p(\overline{X}) \in \mathit{BaseAtom} \\
\mathbf{L}_\Omega[\![p(\overline{X})]\!]\, e\, \Delta & = & \text{let } \Delta_{gi} = \mathsf{gi}(\mathsf{pp}(p(\overline{X}))) \text{ in} \\
 & & \text{let } \mu_i = e(p(\overline{X})) \text{ in} \\
 & & \text{let } \Delta_i = \{\delta_{gi} \overset{\mathsf{comb}^{\mathcal{A}}}{\rightarrow} \mu_i \mid \delta_{gi} \in \Delta_{gi}\} \text{ in} \\
 & & \quad \Delta \cup \Delta_i \\
 & & \qquad\qquad \text{where } p(\overline{X}) \in \mathit{NBCall} \\
\mathbf{L}_\Omega[\![p(\overline{X})]\!]\, e\, \Delta & = & \Delta \\
 & & \qquad\qquad \text{otherwise}
\end{array}
$$

The intended meaning of the resulting optimisation table is similar to the meaning we gave in Definition 13.7, except that now, the optimisation information derived for non base call atoms is also *definite*. If a call description δ of a procedure p is such that there exists a call requirement within p, δ_i^m ($i \in \mathsf{pp}$), for which $\delta \sqsubseteq \delta_i^m$, then the literal at that program point i can *definitely* be optimised, regardless of the nature of that literal. Yet, a verification step is still required in order to check what exactly can be optimised if the literal is a non base call atom.

It can be shown that, *if a fixpoint is reached*, the semantics $\mathcal{S}em_\Omega$ is correct with respect to this intended meaning. But in this setting, we can not guarantee that for each Noetherian domain a fixpoint will always be reached. Either extra restrictions on the domain need to be imposed, or widening operations need to be used (Cousot and Cousot 1992a; Cousot and Cousot 1992c; Zaffanella, Bagnara, and M. Hill 1999; Codish, Heaton, and King 1997).

13.6 CTGC reformulated

We reformulate the reuse analysis underlying the CTGC system using the $\mathcal{S}em_{\omega+}$ formalism. The formalisation of the CTGC system consists of defining the domain for expressing the call requirements of the intended optimisations, correctly de-

fining the base atoms, and initialising the base table, and finally, defining the auxiliary functions on elements of this domain.

Call Requirements Domain In the above presented theory, the abstract domain used to express the goal-independent annotations is assumed to be the same abstract domain used to describe the minimal and call requirements. Yet in the formulation of the CTGC system, we clearly use abstract liveness descriptions as our underlying domain, while we derive reuse information tuples based on it. However, given the fact that the operation mapping reuse information tuples onto the associated abstract liveness call descriptions is an isomorphism, this does not in fact contradict our initial assumption.

As reuse information tuples are easier to determine for expressing the possibilities of reuse at a certain tuple, then its associated call descriptions, it is only natural that we continue to use $\mathcal{RI}$ as the domain to express the minimal and call requirements, while elements of $\mathcal{AL}$ are used for the underlying goal-independent annotations.

As the theoretical ordering in $\mathcal{RI}$ (Definition 10.8) is hard to use in practice, we use Definition 10.10 instead. This probably induces some loss of precision, the extent of which is left for future research.

We will thus formalise the CTGC system in $\mathcal{S}em_{\omega+}$ not by propagating elements from $\mathcal{AL}$ but call requirements expressed as reuse information tuples in $\mathcal{RI}$. Recall that this domain, ordered by $\sqsubseteq_r$ (Definition 10.10), is a complete lattice, with bottom element the empty set, and top element the set of all possible combinations of reuse information tuples over the set of variables of interest.

Base Atoms, Base Table Translating the terminology used for the propagation of reuse information into the current setting, we have that the verification of direct reuses corresponds to the verification of minimal requirements at base atoms, while determining indirect reuses consists of verifying and propagating call requirements at non base call atoms.

Before actually defining the call requirements, we need to make some observations concerning the first formalisation of reuse information tuples and their verification. In Chapter 10 our focus was on a correct handling of modules, and therefore translating reuse tuples to the head variables of the procedures to which they refer. For that purpose, we immediately added all the local information to the reuse information tuples, such that the verification of reuses could simply be performed by comparing the call descriptions of a procedure with these tuples. In this chapter, our formalisation expresses a slightly different behaviour as here we expect that minimal (and call) requirements are purely expressed in terms of the literal to be optimised, assuming that these requirements are compared with the call description at that literal, instead of the call description of the procedure to which the literal belongs. Yet this difference is minimal, and can be easily

overcome given the following observation:

Corollary 13.1 *Consider a deconstruction unification $X \Rightarrow f(Y_1, \ldots, Y_n)$ at a program point (i) in a procedure p with reuse information $R_i = \langle\!\langle \{X^{\overline{\epsilon}}\}, U_i, A_{l,i} \rangle\!\rangle$ where U_i is the local in use set, and $A_{l,i}$ is the local structure sharing set. Let $AL_0 = \langle A_0, L_0 \rangle$ be a call description for procedure p. Then we can observe that:*

$$AL_0 \sharp\, R_i \Leftrightarrow \mathsf{combupdate}^{\mathcal{AL}}(AL_0, \langle A_{l,i}, U_i \rangle) \sharp\, \langle\!\langle \{X^{\overline{\epsilon}}\}, \{\,\}, \{\,\} \rangle\!\rangle$$

This is an immediate consequence of the definition of combupdate (Definition 8.21) based on the improved definition for computing liveness information, c.f. Equation (8.6), the definition of verifying the reuse information, and the facts that $\mathsf{extend}_a(L, \{\,\}) = L$ and $\mathsf{comb}_a(A, \{\,\}) = A$.

This observation allows us to define the minimal requirements for deconstruction unifications to allow reuses purely in terms of the involved deconstruction. The generic condition of reuses at deconstructions thus becomes: let $X \Rightarrow Y$, be a deconstruction unification (leaving the exact term corresponding to Y undetermined here), then the minimal requirement for reusing the top level data structure of X is the reuse tuple $\langle\!\langle \{X^{\overline{\epsilon}}\}, \{\,\}, \{\,\} \rangle\!\rangle$.

The base table therefore consists of only one entry, $X \Rightarrow Y$, with minimal requirement $\langle\!\langle \{X^{\overline{\epsilon}}\}, \{\,\}, \{\,\} \rangle\!\rangle$.

Auxiliary Functions The init function is obviously trivial. Before starting to propagate call requirements expressing reuse we initialise the set of call requirements to the empty set. Thus: $\mathsf{init}^{\mathcal{RI}} = \{\,\}$.

Before defining the pseudo-complement operation we must clearly identify the combination operator w.r.t. which we need to define that pseudo-complement. As was already observed when presenting the minimal requirements for the CTGC system, the approach of the initial formalisation of the propagation of reuse information was to compare call descriptions of procedures to reuse information that was already updated with the local annotations, while in this chapter, we want to compare requirements for optimisation with the result of combining call descriptions with local descriptions. Corollary 13.1 showed that this difference can be overcome for direct reuses, we show that in the same way it also can be overcome for the case of indirect reuses:

Corollary 13.2 *Let $R_q = \langle\!\langle D_q, U_q, A_q \rangle\!\rangle$ be a reuse information tuple characterising the optimisation opportunities within a procedure q with head variables $\{X_1, \ldots, X_n\}$. Let the literal at program point (i) within the definition of a procedure p consist of a call $q(Y_1, \ldots, Y_n)$. Moreover, the local information at (i) consists of the tuple $AL_{l,i} = \langle A_{l,i}, U_i \rangle$. Let AL be a call description for procedure p. Then we have:*

$$\begin{array}{c} AL \sharp\, \langle\!\langle \rho\left(D_q\right), \rho\left(U_q\right) \sqcup_{ad} U_i, \mathsf{comb}_a(A_{l,i}, \rho\left(A_q\right)) \rangle\!\rangle \\ \Updownarrow \\ \mathsf{combupdate}^{\mathcal{AL}}(AL, AL_{l,i}) \sharp \rho\left(R_q\right) \end{array}$$

where $\rho(.)$ is the function mapping all occurrences of the formal head variables of q, i.e., *$X_1, \ldots, X_n$, onto the actual arguments of q in the call,* i.e., *$Y_1, \ldots, Y_n$.*

The correctness of this corollary depends on the same insights as the previous one.

In these corollaries we see that reuse information tuples are compared to the combination of abstract liveness descriptions using the $\mathsf{combupdate}^{\mathcal{AL}}$ operation. Hence, we need to define the operation allowing to compute the most general call description AL, such that AL combined with the local description $AL_{l,i}$ at a program point i where some optimisation might be performed, should remain subsumed by the call or minimal requirement at that program point described by a reuse information tuple R_i: $\mathsf{combupdate}^{\mathcal{AL}}(AL, AL_{l,i})\sharp\ R_i$. In this formulation, we consider that R_i is not already updated with that local information, *i.e.*, with $AL_{l,i}$.

According to Theorem 10.1 for direct reuse, and Theorem 10.2 for the case of indirect reuse, reuse information tuples can be propagated by compacting them w.r.t. to the local information $AL_{l,i}$ available. Hence, it is only natural to try to express the pseudo-complement operation through that operation.

Definition 13.8 (Pseudo-complement in $\mathcal{RI}$) *Let $R = \langle\!\langle D, U, A \rangle\!\rangle$ be a minimal or call requirement expressed in $\mathcal{RI}$ for a program point (i), and $AL_{l,i} = \langle U_i, A_{l,i} \rangle$ be the goal-independent annotation recorded for that program point, then the generalised pseudo-complement of R w.r.t. $AL_{l,i}$, denoted as $AL_{l,i} \overset{\mathsf{combupdate}^{AL}}{\rightarrow} R$, is a new tuple $R' = \mathsf{compact}(R_u, \mathcal{H})$ (Definition 10.4) where*

$$R_u = \langle\!\langle D, U \sqcup_{ad} U_i, \mathsf{comb}_a(A_{l,i}, A) \rangle\!\rangle$$

and where $\mathcal{H}$ are the head variables of the procedure to which program point (i) belongs.

Using this definition, if a call description satisfies the pseudo-complement then the the result of combining that call description with the local description definitely satisfies the initial call requirement (c.f. Theorem 10.1 and Theorem 10.2).

However, the pseudo-complement as defined here may not cover the least upper bound of all the call descriptions satisfying the initial reuse information. Hence, a more precise definition may exist. Yet, finding that exact definition and comparing it to the definition here is left for future work.

Summary To summarise the instantiation of $\mathcal{S}em_\omega$ for the CTGC system:

- While minimal and call requirements are expressed as elements in $\mathcal{RI}$, the underlying goal-independent annotations are in terms of the domain $\mathcal{AL}$. Given the one-to-one link between elements of these domains, this does not violate ur prerequisites for our optimisation derivation system.

- The base atoms consist of all deconstruction unifications $X \Rightarrow Y$, for which the minimal requirement $\langle\!\langle \{X^{\overline{e}}\}, \{\,\}, \{\,\} \rangle\!\rangle$ must be taken into account.
- Finally, we have:

$$\begin{array}{lcl} \mathsf{init}^{\mathcal{RI}} & = & \{\,\} \\ \mathsf{comb}^{\mathcal{RI}} & = & \mathsf{combupdate}^{\mathcal{AC}} \\ \overset{\mathsf{comb}^{\mathcal{RI}}}{\rightarrow} & = & \overset{\mathsf{combupdate}^{\mathcal{AC}}}{\rightarrow} \end{array}$$

13.7 Discussion

In the above semantics we have restricted the language to non-modular programs. In general, we can lift this restriction by computing all the optimisation tables for the modules within a program in a bottom-up way: if a module A depends on a module B, then the optimisation derivation system should first derive the optimisations for the procedures in B, before deriving the optimisations possible within module A. In such cases, we have no loss of precision. Note that the optimisation derivation system is not concerned with the actual version generation. This still requires a separate pass, which becomes indeed more complicated in the presence of modules. Programs with circular dependencies among modules pose the usual problems as ideally the fixpoint iteration should be done over the module graph (Puebla and Hermenegildo 1999b). See also our discussion on Page 251.

We introduced *minimal requirements*. These are call descriptions μ related to literals l, such that for each call description δ, if $\delta \sqsubseteq \mu$, then l can be optimised. This means that we need an abstract domain in which a suitable order relation $\sqsubseteq$ can be expressed for the intended optimisation. A careful design of the abstract domain is therefore necessary.

We generalised the notion of pseudo-complement such that the operation is not limited to the greatest lower bound of the abstract domain used. We expect that for most domains, the combination operation will indeed be the greatest lower bound, but we did not want to limit our formalisation to this operation. One of the reasons was that for the propagation of reuse information tuples, a specialised combination operation is used. In this context, we also want to be able to reason about the *largest* abstract description δ for which the result of combining it with a goal-independent annotation can still correctly be approximated by the minimal requirement for optimisation. Computing this *largest* abstract description is exactly what the pseudo-complement does with respect to $\sqcap$. Therefore, it is natural to generalise this operation to allow other combination operations.

In the present setting we assumed that the abstract domain is such that the (generalised) pseudo-complement of two elements always exists. This means that in Definition 13.5, δ'_c always satisfies the equation $\mathsf{comb}^{\mathcal{A}}(\delta_a, \delta'_c) \sqsubseteq \delta_b$. This imposes a certain restriction on the domains that can be used. Domains in which the

relative pseudo-complements exist are the *Heyting algebras* (King and Lu 2002; Giacobazzi and Scozzari 1998), a particular case of which being the *condensing* domains (Jacobs and Langen 1992; Giacobazzi, Ranzato, and Scozzari). Yet this guarantee may in general not be valid if other combination operations are used than the greatest lower bound.

In our intuitive example, we suggested that approximations of these pseudo-complements can be a good alternative for situations and domains for which the exact pseudo-complement does not exist (which is for example the case for descriptions in $Def_{\perp}$). For these approximations, there are two alternatives. Either one approximates the exact pseudo-complement, say δ^m, *from above* — $\delta^m \sqsubseteq \overline{\delta^m}$, , or *from below* — $\underline{\delta^m} \sqsubseteq \delta^m$. Approximating from above means that there might be some call descriptions, which, when compared to $\overline{\delta^m}$, erroneously suggest that optimisation is possible. This means that even for base atoms, we can not be sure how to interpret the obtained optimisation results. Therefore, this alternative is not interesting. On the other hand, approximating from below respects safety of the results for base atoms, yet, in some cases, some optimisations might not be spotted — this may be the case for the definition of the pseudo-complement given for the derivation of reuse information tuples. Note that such approximation might impose other restrictions on the abstract domain used. Determining exactly which restrictions requires further investigation.

In our optimisation system we have deliberately taken the least upper bound of the individual call requirements within a procedure instead of the greatest lower bound. The advantage of this approach is that we can spot all possible optimisations. The disadvantage is that if the optimisation derivation system spots that a procedure call might be optimisable, we do not know exactly which optimisations are possible, if at all (this may be the case for some abstract domains). The price to pay is that extra checks are necessary during the actual version generation pass. The alternative, the greatest lower bound, has the advantage that if the optimisation derivation spots optimisations, then these optimisations are definitely possible. The disadvantage though is that an optimisation is only identified as such if *all* the optimisations within the called procedure are safe. Hence, a lot of intermediate combinations of optimisations may be missed, which is exactly what we want to avoid. A possible solution to the loss of precision using $\sqcup$ instead of $\sqcap$ is by collecting the call requirements as sets, keeping them distinct, as we described in Section 13.5.

This work is clearly work in progress. And a formal setting for deriving and generating the adequate versions based on the derived over-approximation of optimisation opportunities needs yet to be created. Intuitively, we think that the derived optimisation opportunities do give a good starting point for the versioning problem as it makes the link between possible optimisations and call descriptions for which these optimisations are safe explicit, making it possible to (automatically) recognise “interesting” versions to be generated. Moreover, in our work we

have faced a number of situations where we had to thoroughly search through the code of our benchmarks in order to see why exactly some reuse could not be performed. Having all the optimisation criteria at hand, propagated up the call graph, we expect that such identification process could be made easier. This means that also for the end-user, interesting and understandable feedback could be given about the optimisations that might have failed for her/his particular code.

13.8 Related Work

Program analyses that are part of optimising compilers are always confronted with the issue of version generation: if a predicate can be optimised in several ways, which versions should be generated knowing that not all optimisations are safe in each calling context? Without a priori information about the use of the predicates, there are two possibilities: either the optimising compiler generates all possible combinations of optimised predicates, or the compiler uses some predefined criterion that guides the number of versions that it will produce (a common criterion being that at most two versions are generated per predicate: a genuine, non-optimised version, and a fully optimised version). While the former ensures that the resulting program can make full use of its optimisation potential, the code explosion it requires may make this technique unfeasible in practice. The alternative option limits the code explosion, but it also limits the optimisation possibilities. It is possible to overcome the above problems of code explosion and suboptimality by using information about how some of the predicates are used. Up to now, the most common technique was to build a top-down analysis and version generation process. In such a system, each call to a predicate is described in the abstract domain, and for each such call a different optimised version is generated. Depending on the granularity of the abstract domain, this may still involve the creation of a large number of versions (Vanhoof and Bruynooghe 1999; Leuschel, Martens, and De Schreye 1998; Henglein and Mossin 1994; Puebla and Hermenegildo 1999a). In (Puebla and Hermenegildo 1999a), for example, the analysis and version generation pass is therefore followed by a pruning pass: this pass tries to identify similar versions and merges them, hence reducing the number of versions. All the above mentioned techniques have one aspect in common: each of the versions is identified by one specific call description for which the procedure call is considered safe. If a different call description is encountered, then none of the analyses can safely decide whether it is still safe to call the optimised version or not (without at least a partial new analysis).

In this chapter we propose an optimisation derivation system that relies on information that is a priori collected (in a call independent way), and that allows to identify each single optimisation by a description of the calls for which that optimisation is safe. In case of CTGC deriving that a data structure is dead en-

ables the optimisation of reusing that data structure. Also other existing abstract domains – such as for example the domains used in (Bevemyr 1996; Lindgren, Bevemyr, and Millroth 1995) – could be coupled with our system. This optimisation information gives the version generation pass almost full knowledge about when the optimisations are possible, enabling more sophisticated heuristics for version generation. Also, given a new call to a procedure, it enables the version generation system to simply compare the actual call description with the description of the calls for which a particular optimisation is known to be safe, hence safely identifying the possible optimisations and therefore the version to be used. No separate analysis is required.

The proposed optimisation derivation system is based on generalised pseudo-complements (for mapping individual minimal requirements to the variables appearing in the head atom of the procedure they belong to), and least upper bounds (for collecting call requirements) with the purpose of deriving a single description of when an optimisation is safe. These ideas are very similar to the ideas behind the so-called *backwards analysis* (as opposed to the classical *forward analyses*), a new analysis system introduced by King and Lu (2002). The goal of a backwards analysis is to derive a description of the calls of a procedure for which that procedure is guaranteed not to fail (w.r.t. some criterion such as termination, moding, etc.). The ideas for backwards analysis are successfully applied to the domain of termination inference (Genaim and Codish 2001; Lu and King 2002) as well as pair sharing where descriptions of calls are derived that guarantee the absence of sharing (Lu and King 2004). See also (Howe, King, and Lu 2004) for an overview of backwards analysis applications in logic programming.

Some of the ideas presented here are clearly present in the backwards analysis system. Yet, there are two reasons why this model of analysis does not fully fit our needs. A first reason is that, per predicate, backwards analysis computes information on a program point (i) using information it computed a step earlier for program point $(i+1)$, the program point associated with the *next* literal within the same procedure w.r.t. (i), under the usual left-to-right resolution scheme. This is necessary if all the underlying basic information still needs to be derived. In our setting we want to separate the process of deriving the underlying information from the derivation of the optimisations they allow, which means that we consider that all such information is already present. This also means that we do not need to thread each of the call requirements from one program point to the other, as we can immediately translate the optimisation information at a program point to the head of the predicate definition. Moreover, backwards analysis is based on the relative pseudo-complement. We deliberately decided to generalise this operation as we want to be able to use other combination operators than simply the greatest lower bound of the lattice. Finally, another reason why our goal does not immediately fit within backwards analysis is that if a predicate contains more than one literal that can be optimised, then the call descriptions for each of these

optimisations to occur are conjoined during backwards analysis. This results in a restrictive call description for which *all* optimisations are safe, instead of a description for which some *individual* optimisation may be done. Our goal was to derive the non-restrictive call description for optimisation.

More recently, Gallagher (2004) suggested a new approach to the backwards analysis as formulated by (King and Lu 2002) that might replace the need for special abstract domains and relative pseudo-complements. The idea is to make the effect that calls have on specific program points explicit by constructing the resultants semantics (Gabbrielli, Levi, and Meo 1996) of the original program. This makes the behaviour at these program points of interest observable. Deriving descriptions of calls for which some properties hold at these program points can be obtained by using a standard abstract interpretation framework, and requiring no further restrictions on the abstract domain.
The use of the resultants semantics may be a promising approach for a more clean and compact formulation of the optimisation derivation system developed here.

13.9 Conclusion

Collecting information about possible optimisations within a program *before* actually generating code supporting the optimisations is, in our view, an essential step for a better understanding of which versions need to be generated to obtain better global optimisation results. Here we have developed a mechanism which makes it possible to relate the optimisations with the call descriptions for which they are safe. The novelty of this approach is that the actual version generation pass is totally separated from the characterisation of the possible optimisation (for which one could use a maximally optimising version generator (Puebla and Hermenegildo 1999a), or any other version generator). Another important advantage is that if a new use of a predicate is encountered, no new analysis is needed as it suffices to compare that use with the optimisation requirements derived by our system in order to know what version to generate for that use.

We applied the framework to the derivation of the optimisation opportunities detected by our CTGC system. The instantiation remains purely theoretical, as the implementation needs yet to be done. Such an instantiation would be useful to acknowledge our intuition about the ease of understanding failed reuse possibilities, as well as to verify how version generation systems could be implemented on top of it. All this is a matter of future work.

Chapter 14

Conclusion

The goal of this thesis was to develop a complete compile-time garbage collection system for the pure declarative language Mercury. This involves two key issues: the memory usage of the programs optimised using the CTGC system must be safe, even in the presence of modules — *i.e.*, it is a severe error to reuse or deallocate memory during the execution of a program when the information stored in that memory is still needed by that program — and the CTGC system should be of practical use for real-life programs.

We tackled the first issue by meticulously building the compile-time garbage collection system as a sequence of different program analyses that act upon the original source program. We used a denotational approach to define the *natural semantics* of a Mercury program, and used this platform as a base to express each of the involved program analyses. The correctness of these analyses is proven by systematically relating a *concrete domain* and its operations, to an *abstract domain* and its corresponding operations. The role of the concrete domain is to express the run-time properties one wishes to observe for any given Mercury program, while the carefully designed abstract domain correctly approximates these properties. The properties of interest in the compile-time garbage collection system are: *structure sharing*, *liveness information*, and finally, *reuse information*. To express program analysis in the presence of modules, we defined the *goal-independent semantics* of a Mercury program and proved its conditional equivalence with the natural semantics. By using this semantics instead of the natural semantics, we could correctly transpose each of the program analyses of the CTGC system to a modular setting.

We successfully addressed the second issue by taking care to verify the feasibility and efficiency of the CTGC system at each of its development steps using an appropriate prototype implementation and accompanying set of benchmarks. Ultimately this has led to the incorporation of the CTGC system into an existing Mercury compiler, the Melbourne Mercury Compiler. During the implementa-

tion of that system a number of practical aspects needed to be dealt with: heuristics to choose how dead cells should best be reused, techniques of increasing the overall precision of the analyses, and a widening operation to speed up the analysis when needed. The result is a working CTGC system, ready to be used for even larger programs.

We discuss the specific contributions of this work.

Denotational Semantics We defined a number of new semantics of first-order Mercury programs using a denotational approach. This facilitates the development of new program analyses: it suffices to define a concrete domain and a safe approximating domain w.r.t. that concrete domain in order to guarantee the safeness of the resulting analysis obtained by instantiating the adequate semantics with the abstract domain. This approach was inspired by (Marriott, Søndergaard, and Jones 1994). The novelty resides in the explicit formulation of the semantics for Mercury programs (instead of pure first-order Prolog), and the formal characterisation of the equivalence between the *natural semantics*, $\mathcal{S}em_M$, and the *goal-independent based semantics*, $\mathcal{S}em_{M^\bullet}$. This equivalence is essential for proving the correctness of a program analysis defined in terms of $\mathcal{S}em_{M^\bullet}$ w.r.t. the actual concrete meaning of a program defined as an instantiation of $\mathcal{S}em_M$.

Basic Structure Sharing and Liveness Information Mulkers (1991) developed a liveness analysis for first-order Prolog programs based on structure sharing information. Both analyses are formulated in terms of a generic top-down framework for abstract interpretation (Bruynooghe 1991). We adapted the underlying ideas of these analyses to the context of Mercury programs and the semantics we defined for these programs.

For the structure sharing analysis this required the careful redesign of the concrete and abstract structure sharing domains with which we could prove that the results in a goal-independent setting are equivalent to the results obtained when using the concrete domain in the natural semantics.

Compared to (Mulkers 1991), the novelty of the liveness analysis resides in the non-deterministic execution schemes that it takes into account when approximating the live data structures within a procedure. Previously, only forward execution was considered, relying on an instrumented run-time system to guarantee that reused garbage cells are reset upon backtracking. The main reason why we can approximate the live memory cells in the presence of backtracking is the strictly moded character of the Mercury language.

Reuse Analysis An important contribution of our work is the definition of the *reuse analysis*. This analysis identifies the garbage cells within a program, finds ways of reusing these cells locally, and propagates these reuse results in the call

graph of the program. This reuse analysis is especially challenging in the presence of modules.

Modular Analysis Mercury has an advanced module system where each module is meant to be compiled separately. We adapt each of the previously developed analyses to conform with such a compilation scheme.
For the reuse analysis in particular this consists in adapting the analysis such that it derives for each predicate a *reuse condition* that serves as a pre-condition that must be met by the caller in order for the memory reuses to be safe. We prove that the propagation of these conditions is correct, therefore guaranteeing that no unsafe memory reuse will ever be performed during the execution of a Mercury program.

Optimisation Derivation System In this work we create at most two versions for each analysed procedure: a version with optimised memory usage for which the safeness can only be guaranteed if the caller meets the reuse conditions derived for that procedure, and a plain non-optimised version that is always safe to use. We use this heuristic to limit the number of versions that could otherwise be created for each of the analysed procedures. The drawback is that many opportunities for reuse may be missed.

To overcome this limitation, we formulated a tentative framework to characterise optimisation opportunities — here consisting of possibilities of structure reuse — by the call descriptions for which they can be safe. This framework presents many similarities with *backwards analysis* (King and Lu 2002) which Gallagher (2004) recently linked to the area of *resultants semantics* (Gabbrielli, Levi, and Meo 1996). The latter view simplifies the operations present in backwards analysis and should therefore be profitable for the optimisation derivation framework too.

Working CTGC system Finally, the main contribution of our work is the integration of each of the described analyses into a complete CTGC system embedded within the Melbourne Mercury compiler. To the best of our knowledge, this is the first and only complete CTGC system that has ever been built for a programming language. We experimented with this system using a number of small and medium-sized benchmarks. For some of these benchmarks, memory savings of up to 50% could be observed. These results are especially remarkable as they were obtained with benchmarks that were not specifically fine-tuned for their use with the CTGC system.

In the following paragraphs we outline some interesting guidelines for future work.

Stable CTGC System Currently, the CTGC system is implemented in a separate branch of the Mercury compiler. A natural step is to bring this work to the main branch, and measure the usefulness of the CTGC system on larger projects. A beautiful benchmark would be the Melbourne Mercury Compiler itself. We believe that there should be a high potential for structure reuse, yet these reuses may in a first experiment not be detected.

On a short term, the CTGC system could be upgraded to include the following features.

- *Provide better feedback*. Currently the feedback about the detected reuses and more importantly the *missed* reuses within a Mercury program is limited to obscure messages. This feedback should be more elaborated. As most of the information about structure sharing as well as liveness is at hand, this should be achievable with little effort.
- *Limit the reuses*. Looking at our benchmarks we see that the impact of the structure reuses detected by the CTGC system can be severely limited when this system detects too many opportunities for reuse. Once procedures for which too much structure reuse are (automatically) identified, the programmer should have the possibility of annotating of annotating her/his code to specifically limit the reuses of some of the data structures involved by using dedicated declarations.
- *Automatically limit the reuses*. At a later stage, we can try to have the compiler to limit the reuses automatically. We think that during the development cycle of a program, the compiler can already collect the uses of the procedures defined in it. This information could then be used to automatically determine the reuse opportunities that each of these uses would allow, hence creating only a reuse version incorporating those reuses which are of interest for the overall program. Note that such an approach is completely different from the theoretical approach of the optimisation derivation framework developed in this thesis.
- *Cooperation with the RTGC*. The study of the benchmarks has revealed that the impact of the CTGC system on the execution time of the analysed program is strongly related to the memory behaviour of the program expected by the run-time system. Obviously, in the benchmarks where we observed a decrease in execution speed, the RTGC and CTGC step on each others toes. A closer study of how both systems can be tuned for a better cooperation is mandatory.

Region-based Memory Management In the area of functional programming, *regions* have become a hot topic in the context of automatic memory management (Tofte and Talpin 1997). While traditionally, the memory of a program is split into a stack and one heap, here, the dynamically allocated memory is split over a number of different regions. When all the data stored in a region becomes

dead, the whole region is collected at once, as such removing the overhead of collecting the individual cells. The use of these regions can be in the hands of the programmer (Tofte, Birkedal, Elsman, Hallenberg, Højfeld, Sestoft, and Bertelsen 1997), yet automatic region inference is obviously preferred (Tofte and Birkedal 1998). While first steps have been undertaken to translate the region based memory organisation into the context of Prolog (Makholm 2000), it remains an interesting issue on how CTGC combines with regions.

Other Programming Paradigms Compile-time garbage collection is also an interesting issue for languages such as Java. In (Shaham, Kolodner, and Sagiv 2001) the authors have studied the gap between the moment when an object becomes dead and the moment at which it is collected by the run-time garbage collector, as well as the impact on the general performance of the program when that gap is reduced to a minimum. This has led to a number of insights enabling some form of compile-time garbage collection (Shaham, Yahav, Kolodner, and Sagiv 2003). It would be interesting to see how their findings match with our experience, and how our experience can bring to new insights in the domain of CTGC for object-oriented languages.

Appendix A

Source code: `labelopt`

```
%-----------------------------------------------------------------------%
% Copyright (C) 1994-1997 The University of Melbourne.
% This file may only be copied under the terms of the GNU General
% Public License - see the file COPYING in the Mercury distribution.
%-----------------------------------------------------------------------%

% mylabelopt.m (previously labelopt.m) - module to eliminate
% useless labels and dead code.

% Author: zs.
%
% File adapted for use in the prototype CTGC system by Nancy Mazur. The main
% changes consist of the explicit inclusion of list-manipulating predicates
% while the opt-util related predicates are substituted by dummies.
%-----------------------------------------------------------------------%

:- module mylabelopt.

:- interface.

:- import_module bool, list.
:- import_module llds.

        % Build up a set showing which labels are branched to,
        % then traverse the instruction list removing unnecessary labels.
        % If the instruction before the label branches away, we also
        % remove the instruction block following the label.

:- pred mylabelopt_main(list(instruction), bool, list(instruction), bool).
:- mode mylabelopt_main(in, in, out, out) is det.

        % Build up a set showing which labels are branched to.

:- pred mylabelopt__build_useset(list(instruction), set(label)).
:- mode mylabelopt__build_useset(in, out) is det.

%-----------------------------------------------------------------------%

:- implementation.

:- import_module opt_util.
:- import_module std_util.

mylabelopt_main(Instrs0, Final, Instrs, Mod) :-
        mylabelopt__build_useset(Instrs0, Useset),
        mylabelopt__instr_list(Instrs0, yes, Useset, Instrs1, Mod),
        ( Final = yes, Mod = yes ->
                mylabelopt_main(Instrs1, Final, Instrs, _)
        ;
                Instrs = Instrs1
        ).

%-----------------------------------------------------------------------%

mylabelopt__build_useset(Instrs, Useset) :-
```

```
        set_init(Useset0),
        mylabelopt__build_useset_2(Instrs, Useset0, Useset).

:- pred mylabelopt__build_useset_2(list(instruction), set(label), set(label)).
:- mode mylabelopt__build_useset_2(in, in, out) is det.

mylabelopt__build_useset_2([], Useset, Useset).
mylabelopt__build_useset_2([Instr | Instructions], Useset0, Useset) :-
        Instr = Uinstr - _Comment,
        opt_util_instr_labels(Uinstr, Labels, _CodeAddresses),
        set_insert_list(Useset0, Labels, Useset1),
        mylabelopt__build_useset_2(Instructions, Useset1, Useset).

%-----------------------------------------------------------------------%

        % Go through the given instruction sequence. When we find a label,
        % we check whether the label can be branched to either from within
        % the procedure or from the outside. If yes, we leave it alone.
        % If not, we delete it. We delete the following code as well if
        % the label was preceded by code that cannot fall through.

:- pred mylabelopt__instr_list(list(instruction), bool, set(label),
        list(instruction), bool).
:- mode mylabelopt__instr_list(in, in, in, out, out) is det.

mylabelopt__instr_list([], _Fallthrough, _Useset, [], no).
mylabelopt__instr_list([Instr0 | MoreInstrs0],
                Fallthrough, Useset, MoreInstrs, Mod) :-
        Instr0 = Uinstr0 - _Comment,
        ( Uinstr0 = label(Label) ->
                (
                        ( Label = exported(_)
                        ; Label = local(_)
                        ; set_member(Label, Useset)
                        )
                ->
                        ReplInstrs = [Instr0],
                        Fallthrough1 = yes,
                        Mod0 = no
                ;
                        mylabelopt__eliminate(Instr0, yes(Fallthrough),
                                ReplInstrs, Mod0),
                        Fallthrough1 = Fallthrough
                )
        ;
                ( Fallthrough = yes ->
                        ReplInstrs = [Instr0],
                        Mod0 = no
                ;
                        mylabelopt__eliminate(Instr0, no, ReplInstrs, Mod0)
                ),
                opt_util_can_instr_fall_through(Uinstr0, Canfallthrough),
                ( Canfallthrough = yes ->
                        Fallthrough1 = Fallthrough
                ;
                        Fallthrough1 = no
                )
        ),
        mylabelopt__instr_list(MoreInstrs0, Fallthrough1, Useset,
                MoreInstrs1, Mod1),
        list_append(ReplInstrs, MoreInstrs1, MoreInstrs),
        ( Mod0 = no, Mod1 = no ->
                Mod = no
        ;
                Mod = yes
        ).

        % Instead of removing eliminated instructions from the instruction list,
        % we can replace them by placeholder comments. The original comment
        % field on the instruction is often enough to deduce what the
        % eliminated instruction was.

:- pred mylabelopt__eliminate(instruction, maybe(bool), list(instruction), bool).
:- mode mylabelopt__eliminate(in, in, out, out) is det.

mylabelopt__eliminate(Uinstr0 - Comment0, Label, Instr, Mod) :-
        labelopt_eliminate_total(Total),
        (
                Total = yes,
                Instr = [],
                Mod = yes
        ;
                Total = no,
                ( Uinstr0 = comment(_) ->
```

```
                        Comment = Comment0,
                        Uinstr = Uinstr0,
                        Mod = no
                    ;
                        ( Label = yes(Follow) ->
                            ( Follow = yes ->
                                Uinstr = comment("eliminated label only")
                            ;
                                % Follow = no,
                                Uinstr = comment("eliminated label and block")
                            )
                        ;
                            % Label = no,
                            Uinstr = comment("eliminated instruction")
                        ),
                        Comment = Comment0,
                        Mod = yes
                    ),
                    Instr = [Uinstr - Comment]
            ).

:- pred labelopt_eliminate_total(bool).
:- mode labelopt_eliminate_total(out) is det.

labelopt_eliminate_total(yes).

%-----------------------------------------------------------------------------%
% opt_util-related definitions

        % dummy
:- pred opt_util_instr_labels(instr,list(label),list(code_addr)).
:- mode opt_util_instr_labels(in, out, out) is det.

opt_util_instr_labels(_,[],[]).

        % dummy
:- pred opt_util_can_instr_fall_through(instr, bool).
:- mode opt_util_can_instr_fall_through(in, out) is det.

opt_util_can_instr_fall_through(_,yes).

%-----------------------------------------------------------------------------%
% set-related definitions

        % set(T) == set_ordlist(T) == list(T).
:- type set(T) == list(T).

:- pred set_init(set(T)).
:- mode set_init(out) is det.

set_init([]).

:- pred set_insert_list(set(T),list(T),set(T)).
:- mode set_insert_list(in,in,out) is det.

set_insert_list(Set0,List0,Set):-
        list_sort_and_remove_dups(List0,List),
        list_merge_and_remove_dups(List,Set0,Set).

:- pred set_member(T,set(T)).
:- mode set_member(in,in) is semidet.

set_member(T,L):- list_member(T,L).

%-----------------------------------------------------------------------------%
% list-related definitions

:- pred list_append(list(T),list(T),list(T)).
:- mode list_append(in,in,out) is det.

list_append([],Y,Y).
list_append([X|Xs],Y,[X|Zs]):-
        list_append(Xs,Y,Zs).

:- pred list_sort_and_remove_dups(list(T),list(T)).
:- mode list_sort_and_remove_dups(in,out) is det.

list_sort_and_remove_dups(L0, L):-
        list_merge_sort(L0, L1),
        list_remove_adjacent_dups(L1, L).

:- pred list_merge_sort(list(T),list(T)).
:- mode list_merge_sort(in,out) is det.
```

```
list_merge_sort([], []).
list_merge_sort([X], [X]).
list_merge_sort(List, SortedList) :-
        List = [_,_|_],
        list_length(List, Length),
        HalfLength is Length // 2,
        ( list_split_list(HalfLength, List, Front, Back) ->
                list_merge_sort(Front, SortedFront),
                list_merge_sort(Back, SortedBack),
                list_merge(SortedFront, SortedBack, SortedList)
        ;
                error("list__merge_sort")
        ).

:- pred list_length(list(T),int).
:- mode list_length(in,out) is det.

list_length(L, N) :-
        list_length_2(L, 0, N).

:- pred list_length_2(list(T), int, int).
:- mode list_length_2(in, in, out) is det.

list_length_2([], N, N).
list_length_2([_ | L1], N0, N) :-
        N1 is N0 + 1,
        list_length_2(L1, N1, N).

:- pred list_split_list(int, list(T), list(T), list(T)).
:- mode list_split_list(in, in, out, out) is semidet.

list_split_list(N, List, Start, End) :-
        ( N = 0 ->
                Start = [],
                End = List
        ;
                N > 0,
                N1 is N - 1,
                N1 = 1,
                List = [Head | List1],
                Start = [Head | Start1],
                list_split_list(N1, List1, Start1, End)
        ).

:- pred list_merge(list(T), list(T), list(T)).
:- mode list_merge(in, in, out) is det.

list_merge(A, B, C) :-
        ( A = [X|Xs] ->
                ( B = [Y|Ys] ->
                        C = [Z|Zs],
                        (
                            compare(<, X, Y)
                        ->
                            Z = X,
                            list_merge(Xs, B, Zs)
                        ;
                            Z = Y,
                            list_merge(A, Ys, Zs)
                        )
                ;
                        C = A
                )
        ;
                C = B
        ).

:- pred list_remove_adjacent_dups(list(T),list(T)).
:- mode list_remove_adjacent_dups(in,out) is det.

list_remove_adjacent_dups([], []).
list_remove_adjacent_dups([X|Xs], L) :-
        list_remove_adjacent_dups_2(Xs, X, L).

:- pred list_remove_adjacent_dups_2(list(T), T, list(T)).
:- mode list_remove_adjacent_dups_2(in, in, out) is det.

list_remove_adjacent_dups_2([], X, [X]).
list_remove_adjacent_dups_2([X1|Xs], X0, L) :-
        (
            X0 = X1
        ->
            list_remove_adjacent_dups_2(Xs, X1, L)
        ;
```

```
            L = [X0 | L0],
            list_remove_adjacent_dups_2(Xs, X1, L0)
        ).


:- pred list_merge_and_remove_dups(list(T),list(T),list(T)).
:- mode list_merge_and_remove_dups(in,in,out) is det.

list_merge_and_remove_dups(A, B, C) :-
        list_merge(A,B,C).

:- pred list_member(T,list(T)).
:- mode list_member(in,in) is semidet.

list_member(X, [X | _]).
list_member(X, [_ | Xs]) :-
        list_member(X, Xs).
```

Appendix B

Details of the ICFP2000 benchmark

This appendix presents the details of the memory usage and timings of the processing of each of the individual scenes for each of the CTGC-configurations occurring in Table 12.3 (page 270). Recall that each of these configurations is compiled without reuse enabled in the basic libraries of the compiler.

	No Reuse		graph match -		graph match cc		graph samecons -		graph samecons cc	
input	nr_m (kWord)	nr_t (sec)	1_m %	1_t %	2_m %	2_t %	3_m %	3_t %	4_m %	4_t %
checked-cone.gml	3323.51	0.24	-6.97	-8.33	-15.03	4.17	-5.11	-4.17	-13.80	12.50
checked-cube.gml	4006.85	0.26	-6.16	-3.85	-12.61	7.69	-4.50	-3.85	-11.42	19.23
checked-cylinder.gml	7693.15	0.44	-8.99	-4.55	-13.79	9.09	-6.57	-2.27	-12.00	18.18
checked-sphere.gml	6078.05	0.36	-12.32	-5.56	-17.89	5.56	-9.70	2.78	-15.93	19.44
cylinder.gml	25112.53	2.11	-27.54	-10.90	-39.02	-1.90	-25.43	-8.06	-37.44	1.42
dice.gml	542412.75	50.98	-29.34	1.41	**-57.64**	6.90	-28.68	3.24	**-57.21**	5.90
fib.gml	39760.55	3.40	-29.84	-6.76	**-72.10**	-0.88	-29.00	-4.71	**-71.42**	-1.18
golf.gml	44127.04	3.40	-10.75	10.59	-24.18	16.18	-9.63	13.53	-23.72	20.88
mcapsule.gml	4517.83	0.38	-19.01	-7.89	-32.45	2.63	-16.32	-5.26	-30.12	7.89
mintersect.gml	3006.21	0.26	-16.46	0.00	-32.81	7.69	-14.41	0.00	-30.85	15.38
mtest1.gml	3809.94	0.30	-22.10	-10.00	-31.86	0.00	-18.67	-6.67	-29.23	3.33
mtest10.gml	27672.61	2.05	-21.40	-6.83	-40.87	1.95	-19.57	-5.37	-39.18	4.39
mtest11.gml	26345.10	2.03	-7.67	-2.46	-20.89	8.87	-6.40	-1.48	-20.16	12.81
mtest2.gml	1256.88	0.13	-18.77	-7.69	-37.28	7.69	-18.67	0.00	-37.22	7.69
mtest3.gml	3870.94	0.30	-21.23	-10.00	-30.84	0.00	-17.61	-6.67	-28.00	6.67
mtest4.gml	6935.63	0.53	-24.78	-9.43	-39.58	1.89	-21.38	-3.77	-36.92	9.43
mtest5.gml	8720.56	0.65	-24.72	-9.23	-38.77	1.54	-21.15	-3.08	-36.03	9.23
mtest6.gml	11165.26	0.83	-27.34	3.61	**-49.47**	15.66	-24.61	8.43	**-47.31**	15.66
mtest7.gml	120409.14	8.68	-23.04	12.21	**-47.88**	17.05	-20.14	-3.34	**-45.52**	18.78
mtest8.gml	3006.19	0.26	-16.46	0.00	-32.81	11.54	-14.41	0.00	-30.85	15.38
mtest9.gml	5607.97	0.43	-27.38	-9.30	-42.56	-2.33	-24.21	-4.65	-39.81	4.65
munion.gml	5388.78	0.46	-18.74	-8.70	-31.78	0.00	-15.90	-6.52	-29.58	4.35
reflect.gml	38609.22	2.92	-21.88	-7.53	-39.27	5.82	-18.33	-6.51	-36.75	6.16
reflect2.gml	39375.85	2.92	-21.84	-7.19	-39.26	8.22	-18.28	-4.45	-36.73	8.56
spheres.gml	13101.37	0.95	-24.56	-5.26	-39.95	2.11	-22.40	-5.26	-39.01	8.42
spheres2.gml	13596.32	0.99	-23.93	-5.05	-39.03	4.04	-21.58	-4.04	-37.86	9.09
spotlight.gml	15444.67	1.27	-33.57	-14.96	-44.52	-5.51	-32.08	-7.09	-43.77	-4.72
total	1024354.9	87.53	-25.60	0.50	**-49.74**	7.31	-24.13	0.86	**-48.69**	7.87

	No Reuse		graph within1 -		graph within1 cc		graph within2 -		graph within2 cc	
input	nr_m (kWord)	nr_t (sec)	5_m %	5_t %	6_m %	6_t %	7_m %	7_t %	8_m %	8_t %
checked-cone.gml	3323.51	0.24	-9.66	-8.33	-16.07	0.00	-9.66	-8.33	-16.07	0.00
checked-cube.gml	4006.85	0.26	-11.90	-7.69	-11.92	3.85	-11.90	-7.69	-11.92	7.69
checked-cylinder.gml	7693.15	0.44	-14.31	-4.55	-14.32	4.55	-14.31	-6.82	-14.32	4.55
checked-sphere.gml	6078.05	0.36	-14.39	-5.56	-14.40	5.56	-14.39	-8.33	-14.40	5.56
cylinder.gml	25112.53	2.11	-32.17	-12.32	-38.87	-2.84	-32.17	-12.32	-38.87	-3.32
dice.gml	542412.75	50.98	-30.62	0.27	-32.68	9.00	-30.62	0.16	-32.68	9.00
fib.gml	39760.55	3.40	-30.43	-7.35	**-49.88**	-0.29	-30.43	-7.35	**-49.88**	-0.29
golf.gml	44127.04	3.40	-17.13	8.53	-18.79	18.82	-17.13	8.53	-18.79	19.12
mcapsule.gml	4517.83	0.38	-24.90	-7.89	-26.11	-2.63	-24.90	-7.89	-26.11	0.00
mintersect.gml	3006.21	0.26	-23.08	-3.85	-23.52	7.69	-23.08	-3.85	-23.52	7.69
mtest1.gml	3809.94	0.30	-24.73	-10.00	-24.75	0.00	-24.73	-10.00	-24.75	-3.33
mtest10.gml	27672.61	2.05	-26.50	-8.29	-32.44	1.46	-26.50	-8.29	-32.44	1.46
mtest11.gml	26345.10	2.03	-16.55	-3.94	-17.02	22.17	-16.55	-3.45	-17.02	22.66
mtest2.gml	1256.88	0.13	-18.84	0.00	-18.92	0.00	-18.84	0.00	-18.92	0.00
mtest3.gml	3870.94	0.30	-23.95	-6.67	-23.98	0.00	-23.95	-6.67	-23.98	0.00
mtest4.gml	6935.63	0.53	-27.33	-9.43	-30.61	0.00	-27.33	-9.43	-30.61	0.00
mtest5.gml	8720.56	0.65	-27.46	-10.77	-30.77	0.00	-27.46	-9.23	-30.77	0.00
mtest6.gml	11165.26	0.83	-29.35	2.41	-37.14	15.66	-29.35	3.61	-37.14	15.66
mtest7.gml	120409.14	8.68	-25.32	1.15	-42.40	18.55	-25.32	1.61	-42.40	19.01
mtest8.gml	3006.19	0.26	-23.08	-3.85	-23.52	7.69	-23.08	0.00	-23.52	7.69
mtest9.gml	5607.97	0.43	-28.82	-9.30	-33.41	-2.33	-28.82	-9.30	-33.41	-2.33
munion.gml	5388.78	0.46	-24.67	-8.70	-24.98	-4.35	-24.67	-8.70	-24.98	-4.35
reflect.gml	38609.22	2.92	-24.12	-9.59	-33.46	5.48	-24.12	-9.59	-33.46	5.48
reflect2.gml	39375.85	2.92	-24.09	-8.90	-33.50	7.88	-24.09	-8.56	-33.50	8.56
spheres.gml	13101.37	0.95	-25.02	-7.37	-29.97	2.11	-25.02	-6.32	-29.97	3.16
spheres2.gml	13596.32	0.99	-24.91	-7.07	-29.68	3.03	-24.91	-7.07	-29.68	3.03
spotlight.gml	15444.67	1.27	-35.06	-13.39	-44.02	-5.51	-35.06	-14.17	-44.02	-5.51
total	1024354.9	87.53	-27.80	-1.65	-33.23	8.91	-27.80	-1.63	-33.23	9.01

	No Reuse		lifo match -		lifo match cc		lifo samecons -		lifo samecons cc	
input	nr_m (kWord)	nr_t (sec)	9_m %	9_t %	10_m %	10_t %	11_m %	11_t %	12_m %	12_t %
checked-cone.gml	3323.51	0.24	-7.59	-8.33	-15.65	0.00	-5.73	-4.17	-14.41	12.50
checked-cube.gml	4006.85	0.26	-6.60	-3.85	-13.06	3.85	-4.95	-3.85	-11.86	19.23
checked-cylinder.gml	7693.15	0.44	-9.61	-4.55	-14.41	6.82	-7.20	0.00	-12.62	22.73
checked-sphere.gml	6078.05	0.36	-12.97	-5.56	-18.55	5.56	-10.36	-2.78	-16.59	19.44
cylinder.gml	25112.53	2.11	-27.53	-8.53	-39.01	-2.84	-25.41	-8.06	-37.42	2.84
dice.gml	542412.75	50.98	-29.14	1.92	**-57.44**	6.47	-28.49	1.65	**-57.01**	5.88
fib.gml	39760.55	3.40	-30.00	-6.18	**-72.26**	-1.76	-29.15	-7.06	**-71.58**	-0.59
golf.gml	44127.04	3.40	-10.53	11.18	-23.96	17.35	-9.41	12.65	-23.50	24.12
mcapsule.gml	4517.83	0.38	-19.37	-5.26	-32.81	0.00	-16.68	-5.26	-30.48	7.89
mintersect.gml	3006.21	0.26	-16.55	0.00	-32.90	7.69	-14.49	0.00	-30.94	11.54
mtest1.gml	3809.94	0.30	-22.89	-6.67	-32.65	0.00	-19.46	-6.67	-30.02	6.67
mtest10.gml	27672.61	2.05	-21.53	-6.83	-41.00	2.93	-19.70	-6.34	-39.31	4.39
mtest11.gml	26345.10	2.03	-7.67	-0.49	-20.89	7.88	-6.40	-2.46	-20.16	15.76
mtest2.gml	1256.88	0.13	-18.79	-7.69	-37.30	0.00	-18.69	-7.69	-37.24	7.69
mtest3.gml	3870.94	0.30	-22.01	-6.67	-31.62	0.00	-18.38	-6.67	-28.77	6.67
mtest4.gml	6935.63	0.53	-25.51	-5.66	-40.31	-1.89	-22.11	-3.77	-37.65	11.32
mtest5.gml	8720.56	0.65	-25.54	-4.62	-39.59	-1.54	-21.97	-4.62	-36.84	12.31
mtest6.gml	11165.26	0.83	-27.90	6.02	**-50.03**	13.25	-25.17	7.23	**-47.88**	16.87
mtest7.gml	120409.14	8.68	-21.86	-4.26	**-46.69**	16.13	-18.95	-4.49	-44.33	20.51
mtest8.gml	3006.19	0.26	-16.55	0.00	-32.90	11.54	-14.49	0.00	-30.94	11.54
mtest9.gml	5607.97	0.43	-27.79	-6.98	-42.97	-2.33	-24.62	-4.65	-40.22	6.98
munion.gml	5388.78	0.46	-19.38	-6.52	-32.42	0.00	-16.54	-6.52	-30.22	6.52
reflect.gml	38609.22	2.92	-20.55	-5.14	-37.94	1.71	-17.01	-5.82	-35.43	8.56
reflect2.gml	39375.85	2.92	-20.52	-4.11	-37.94	4.11	-16.96	-3.42	-35.41	10.62
spheres.gml	13101.37	0.95	-24.76	-1.05	-40.15	2.11	-22.60	-6.32	-39.22	14.74
spheres2.gml	13596.32	0.99	-24.13	0.00	-39.23	3.03	-21.78	-5.05	-38.06	15.15
spotlight.gml	15444.67	1.27	-33.57	-9.45	-44.52	-9.45	-32.08	-10.24	-43.77	-0.79
total	1024354.9	87.53	-25.30	-0.18	**-49.44**	6.49	-23.83	-0.41	**-48.39**	8.71

	No Reuse		lifo within1 -		lifo within1 cc		lifo within2 -		lifo within2 cc	
input	nr_m (kWord)	nr_t (sec)	13_m %	13_t %	14_m %	14_t %	15_m %	15_t %	16_m %	16_t %
checked-cone.gml	3323.51	0.24	-10.09	-4.17	-16.51	4.17	-8.17	-4.17	-14.59	0.00
checked-cube.gml	4006.85	0.26	-8.70	-3.85	-8.72	7.69	-6.90	-3.85	-6.92	7.69
checked-cylinder.gml	7693.15	0.44	-12.65	-2.27	-12.66	9.09	-9.99	-2.27	-10.00	6.82
checked-sphere.gml	6078.05	0.36	-14.07	-5.56	-14.08	8.33	-7.31	0.00	-7.32	8.33
cylinder.gml	25112.53	2.11	-29.70	-9.00	-36.40	-2.84	-16.71	-6.64	-23.41	2.37
dice.gml	542412.75	50.98	-29.57	1.90	-31.62	10.69	-3.56	11.34	-5.61	19.64
fib.gml	39760.55	3.40	-30.26	-8.24	**-49.72**	-1.47	-6.78	-2.65	-26.23	5.59
golf.gml	44127.04	3.40	-11.02	10.29	-12.67	22.35	-7.32	15.59	-8.97	25.59
mcapsule.gml	4517.83	0.38	-20.55	-5.26	-21.75	2.63	-10.45	-5.26	-11.66	7.89
mintersect.gml	3006.21	0.26	-17.09	0.00	-17.53	7.69	-6.21	0.00	-6.65	15.38
mtest1.gml	3809.94	0.30	-24.20	-3.33	-24.23	0.00	-11.85	0.00	-11.87	3.33
mtest10.gml	27672.61	2.05	-23.39	-7.80	-29.37	6.34	-11.81	-3.90	-17.79	10.24
mtest11.gml	26345.10	2.03	-8.80	-0.49	-9.27	12.32	-7.07	-0.49	-7.54	12.81
mtest2.gml	1256.88	0.13	-18.82	0.00	-18.91	0.00	-0.33	0.00	-0.42	7.69
mtest3.gml	3870.94	0.30	-23.31	-6.67	-23.34	0.00	-11.14	-3.33	-11.17	3.33
mtest4.gml	6935.63	0.53	-26.73	-5.66	-30.01	0.00	-12.43	-1.89	-15.71	3.77
mtest5.gml	8720.56	0.65	-26.90	-6.15	-30.21	0.00	-13.41	-1.54	-16.73	3.08
mtest6.gml	11165.26	0.83	-28.86	6.02	-36.65	15.66	-10.74	12.05	-18.53	20.48
mtest7.gml	120409.14	8.68	-22.78	-3.92	-39.85	18.66	-12.69	-0.81	-29.76	23.96
mtest8.gml	3006.19	0.26	-17.09	-3.85	-17.53	7.69	-6.21	0.00	-6.65	15.38
mtest9.gml	5607.97	0.43	-27.79	-9.30	-32.38	0.00	-13.00	-4.65	-17.59	2.33
munion.gml	5388.78	0.46	-20.60	-6.52	-20.91	0.00	-11.01	-4.35	-11.33	4.35
reflect.gml	38609.22	2.92	-22.27	-9.25	-31.61	3.77	-13.30	-2.40	-22.64	8.22
reflect2.gml	39375.85	2.92	-22.23	-7.53	-31.64	5.82	-13.27	-2.05	-22.68	11.30
spheres.gml	13101.37	0.95	-24.76	-2.11	-29.72	5.26	-12.42	1.05	-17.37	6.32
spheres2.gml	13596.32	0.99	-24.57	-1.01	-29.35	4.04	-12.59	2.02	-17.36	7.07
spotlight.gml	15444.67	1.27	-34.81	-10.24	-43.77	-8.66	-15.67	-7.09	-24.63	-1.57
total	1024354.9	87.53	-26.05	-0.58	-31.48	9.87	-7.22	6.52	-12.65	16.92

	No Reuse		random match - cc		random match cc		random samecons -		random samecons cc	
input	nr_m (kWord)	nr_t (sec)	17_m %	17_t %	18_m %	18_t %	19_m %	19_t %	20_m %	20_t %
checked-cone.gml	3323.51	0.24	-6.97	-4.17	-15.03	0.00	-5.11	-4.17	-13.80	12.50
checked-cube.gml	4006.85	0.26	-6.16	-7.69	-12.61	3.85	-4.50	-3.85	-11.42	19.23
checked-cylinder.gml	7693.15	0.44	-8.99	-4.55	-13.79	6.82	-6.57	-2.27	-12.00	18.18
checked-sphere.gml	6078.05	0.36	-12.32	-2.78	-17.89	5.56	-9.70	-2.78	-15.93	16.67
cylinder.gml	25112.53	2.11	-27.07	-8.53	-38.55	-3.32	-24.95	-8.06	-36.96	2.37
dice.gml	542412.75	50.98	-28.92	2.28	**-57.23**	6.14	-28.27	1.24	**-56.79**	5.71
fib.gml	39760.55	3.40	-29.84	-5.88	**-72.10**	-1.76	-29.00	-8.53	**-71.42**	-2.35
golf.gml	44127.04	3.40	-9.88	12.35	-23.31	17.94	-8.76	12.65	-22.85	25.00
mcapsule.gml	4517.83	0.38	-19.01	-5.26	-32.45	0.00	-16.32	-5.26	-30.12	7.89
mintersect.gml	3006.21	0.26	-16.46	0.00	-32.81	7.69	-14.41	0.00	-30.85	15.38
mtest1.gml	3809.94	0.30	-22.10	-3.33	-31.86	0.00	-18.67	-10.00	-29.23	6.67
mtest10.gml	27672.61	2.05	-21.40	-6.83	-40.87	2.44	-19.57	-6.34	-39.18	4.39
mtest11.gml	26345.10	2.03	-7.19	-0.99	-20.41	7.39	-5.92	-1.48	-19.67	15.27
mtest2.gml	1256.88	0.13	-18.77	0.00	-37.28	0.00	-18.67	0.00	-37.22	7.69
mtest3.gml	3870.94	0.30	-21.23	-6.67	-30.84	0.00	-17.61	-6.67	-28.00	6.67
mtest4.gml	6935.63	0.53	-24.78	-5.66	-39.58	-1.89	-21.38	-5.66	-36.92	11.32
mtest5.gml	8720.56	0.65	-24.72	-6.15	-38.77	0.00	-21.15	-4.62	-36.03	12.31
mtest6.gml	11165.26	0.83	-27.34	6.02	**-49.47**	13.25	-24.61	7.23	**-47.31**	18.07
mtest7.gml	120409.14	8.68	-21.34	-4.38	**-46.17**	16.82	-18.43	-4.49	-43.81	20.16
mtest8.gml	3006.19	0.26	-16.46	0.00	-32.81	7.69	-14.41	0.00	-30.85	11.54
mtest9.gml	5607.97	0.43	-27.38	-9.30	-42.56	-2.33	-24.21	-6.98	-39.81	6.98
munion.gml	5388.78	0.46	-18.74	-6.52	-31.78	0.00	-15.90	-6.52	-29.58	6.52
reflect.gml	38609.22	2.92	-19.53	-5.14	-36.92	2.05	-15.98	-6.16	-34.40	8.22
reflect2.gml	39375.85	2.92	-19.49	-3.42	-36.92	4.45	-15.94	-4.45	-34.39	10.62
spheres.gml	13101.37	0.95	-23.54	0.00	-38.93	2.11	-21.38	-6.32	-38.00	14.74
spheres2.gml	13596.32	0.99	-22.95	-1.01	-38.05	2.02	-20.60	-5.05	-36.88	15.15
spotlight.gml	15444.67	1.27	-32.82	-9.45	-43.77	-9.45	-31.33	-9.45	-43.02	-1.57
total	1024354.9	87.53	-24.90	0.09	**-49.04**	6.36	-23.43	-0.75	**-47.99**	8.48

	No Reuse		random within1 -		random within1 cc		random within2 -		random within2 cc	
input	nr_m (kWord)	nr_t (sec)	21_m %	21_t %	22_m %	22_t %	23_m %	23_t %	24_m %	24_t %
checked-cone.gml	3323.51	0.24	-9.48	-4.17	-15.89	4.17	-7.55	-4.17	-13.97	8.33
checked-cube.gml	4006.85	0.26	-8.25	-3.85	-8.27	7.69	-6.45	-3.85	-6.47	11.54
checked-cylinder.gml	7693.15	0.44	-12.03	0.00	-12.04	6.82	-9.36	-4.55	-9.37	13.64
checked-sphere.gml	6078.05	0.36	-13.41	0.00	-13.42	5.56	-6.66	-2.78	-6.67	13.89
cylinder.gml	25112.53	2.11	-29.24	-5.69	-35.94	-0.47	-16.30	-5.69	-22.99	4.74
dice.gml	542412.75	50.98	-29.35	6.14	-31.40	10.26	-3.17	15.56	-5.22	20.26
fib.gml	39760.55	3.40	-30.10	-7.06	**-49.56**	-1.76	-6.60	-1.47	-26.06	7.06
golf.gml	44127.04	3.40	-10.37	3.24	-12.02	24.12	-6.78	-2.06	-8.43	28.24
mcapsule.gml	4517.83	0.38	-20.19	-2.63	-21.39	5.26	-10.21	-5.26	-11.42	10.53
mintersect.gml	3006.21	0.26	-17.00	0.00	-17.44	11.54	-6.15	-3.85	-6.59	19.23
mtest1.gml	3809.94	0.30	-23.41	-6.67	-23.44	-3.33	-11.06	-3.33	-11.08	6.67
mtest10.gml	27672.61	2.05	-23.26	-5.37	-29.24	5.85	-10.92	-2.44	-16.90	9.27
mtest11.gml	26345.10	2.03	-8.32	2.96	-8.79	12.32	-6.63	-0.49	-7.10	12.32
mtest2.gml	1256.88	0.13	-18.80	-7.69	-18.89	0.00	-0.31	0.00	-0.40	7.69
mtest3.gml	3870.94	0.30	-22.53	-6.67	-22.56	0.00	-10.37	-6.67	-10.39	6.67
mtest4.gml	6935.63	0.53	-26.00	0.00	-29.28	0.00	-11.93	-3.77	-15.21	7.55
mtest5.gml	8720.56	0.65	-26.09	0.00	-29.40	0.00	-12.65	-4.62	-15.96	6.15
mtest6.gml	11165.26	0.83	-28.30	7.23	-36.09	14.46	-10.00	2.41	-17.79	24.10
mtest7.gml	120409.14	8.68	-22.26	-0.12	-39.33	10.14	-11.25	0.81	-28.32	22.24
mtest8.gml	3006.19	0.26	-17.00	0.00	-17.44	11.54	-6.15	-3.85	-6.59	19.23
mtest9.gml	5607.97	0.43	-27.38	-4.65	-31.97	0.00	-11.19	-6.98	-15.78	6.98
munion.gml	5388.78	0.46	-19.96	-2.17	-20.27	4.35	-10.57	-6.52	-10.89	6.52
reflect.gml	38609.22	2.92	-21.24	-0.34	-30.58	6.85	-11.30	-1.37	-20.65	9.93
reflect2.gml	39375.85	2.92	-21.21	1.71	-30.61	8.90	-11.25	0.68	-20.66	12.67
spheres.gml	13101.37	0.95	-23.54	4.21	-28.50	3.16	-11.53	-4.21	-16.49	8.42
spheres2.gml	13596.32	0.99	-23.40	5.05	-28.17	2.02	-11.70	-2.02	-16.48	8.08
spotlight.gml	15444.67	1.27	-34.07	-8.66	-43.02	-7.87	-14.92	-7.87	-23.88	0.79
total	1024354.9	87.53	-25.65	3.19	-31.08	9.06	-6.53	8.35	-11.96	17.73

References

Allison, L. (1986). *A Practical Introduction to Denotational Semantics*. Cambridge Computer Science Texts. Cambridge-University-Press.

Bagnara, R., P. M. Hill, and E. Zaffanella (2000). Efficient structural information analysis for real CLP languages. In M. Parigot and A. Voronkov (Eds.), *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, Volume 1955 of *Lecture Notes in Computer Science*, Reunion Island, France, pp. 189–206. Springer-Verlag.

Bagnara, R., E. Zaffanella, and P. M. Hill (2000). Enhanced sharing analysis techniques: A comprehensive evaluation. In M. Gabbrielli and F. Pfenning (Eds.), *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Montreal, Canada, pp. 103–114. ACM Press.

Bagnara, R., E. Zaffanella, and P. M. Hill (2005, January). Enhanced sharing analysis techniques: A comprehensive evaluation. *Theory and Practice of Logic Programming 5*(1-2). To appear.

Barbuti, R., R. Giacobazzi, and G. Levi (1993). A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems 15*(1), 133–181.

Bekkers, Y. and P. Tarau (1995, December 4–7). Monadic constructs for logic programming. In J. Lloyd (Ed.), *Proceedings of the International Symposium on Logic Programming*, Cambridge, pp. 51–65. MIT Press.

Bevemyr, J. (1996). *Data-parallel Implementation of Prolog*. Ph. D. thesis, Computing Science Department, Uppsala University, Uppsala, Sweden.

Blanchet, B. (1998, June). Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, pp. 25–37. ACM Press.

Blanchet, B. (1999, October). Escape analysis for object oriented languages: Application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, Volume 34(10) of *ACM SIGPLAN Notices*, Denver, CO, pp. 20–34. ACM Press.

Boehm, H.-J. and M. Weiser (1988). Garbage collection in an uncooperative environment. *Software Practice and Experience 18*(9), 807–820.

Bruynooghe, M. (1991, February). A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming 10*(2), 91–124.

Bruynooghe, M., M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof (2003, May). Termination analysis through combination of type based norms. Technical report, Department of Computer Science; Ben-Gurion University.

Bruynooghe, M., M. Codish, S. Genaim, and W. Vanhoof (2002, September). Reuse of results in termination analysis of typed logic programs. In M. Hermenegildo and G. Puebla (Eds.), *Proceedings of The 9th International Static Analysis Symposium (SAS 2002)*, Volume 2477 of *Lecture Notes in Computer Science*, pp. 477–492. Springer-Verlag.

Bruynooghe, M. and D. De Schreye (1988). Tutorial on abstract interpretation of logic programs. In R. A. Kowalski and K. A. Bowen (Eds.), *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seatle. ALP, IEEE: The MIT Press.

Bruynooghe, M., G. Janssens, and A. Kågedal (1997). Live-structure analysis for logic programming languages with declarations. In L. Naish (Ed.), *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, Leuven, Belgium, pp. 33–47. MIT Press.

Bruynooghe, M. and K. Lau (Eds.) (2004). *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation, Revised Selected Papers*, Volume 3018 of *Lecture Notes in Computer Science*. Springer-Verlag. To appear.

Bueno, F., M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. J. Stuckey (2001, July). A model for inter-module analysis and optimizing compilation. In K.-K. Lau (Ed.), *Tenth International Workshop on Logic-based Program Synthesis and Transformation*, Volume 2042 of *Lecture Notes in Computer Science*, London, UK, pp. 86–102. Springer-Verlag.

Codish, M. and B. Demoen (1993, October). Analysing logic programs using "Prop"-ositional logic programs and a Magic Wand. In D. Miller (Ed.), *Proceedings of the 1993 International Logic Programming Symposium*, Vancouver, pp. 114–129. MIT Press.

Codish, M., M. García de la Banda, M. Bruynooghe, and M. Hermenegildo (1997). Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming 32*(3), 247–261.

Codish, M., S. Genaim, M. Bruynooghe, J. Gallagher, and W. Vanhoof (2003, June). One loop at a time. In *6th International Workshop on Termination (WST 2003)*.

Codish, M., A. Heaton, and A. King (1997, December). Widening Pos for efficient and scalable groundness analysis of logic programs. Technical Report 16-97, University of Kent at Canterbury.

Codish, M., K. Marriott, and C. Taboch (2000). Improving program analyses by structure untupling. *Journal of Logic Programming 43*(3), 251–263.

Conway, T., F. Henderson, and Z. Somogyi (1995). Code generation for Mercury. In J. Lloyd (Ed.), *Logic Programming, Proceedings of the 1995 International Symposium (ILPS'95)*, Portland, Oregon, pp. 242–256. MIT Press.

Cortesi, A. and G. Filé (1991). Abstract interpretation of logic programs: An abstract domain for groundness, sharing, freeness and compoundness analysis. In P. Hudak

and N. Jones (Eds.), *Proc. Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPMA'91*, Connecticut,USA, pp. 52–61. SIGPLAN Notices vol. 26 nb. 11.

Cortesi, A., G. Filé, and W. Winsborough (1991). Prop revisited: Propositional formula as abstract domain for groundness analysis. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pp. 322–327.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, pp. 238–252.

Cousot, P. and R. Cousot (1992a, May/July). Abstract interpretation and application to logic programs. *Journal of Logic Programming 13*(2 & 3), 103–179.

Cousot, P. and R. Cousot (1992b). Abstract interpretation frameworks. *Journal of Logic Programming 2*(4), 511–547.

Cousot, P. and R. Cousot (1992c). Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing (Eds.), *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, Volume 631 of *Lecture Notes in Computer Science*, Leuven, Belgium, pp. 269–295. Springer-Verlag.

Davey, B. A. and H. A. Priestley (2002). *An Introduction to Lattices and Order, 2nd Edition*. Cambridge: Cambridge University Press.

Debray, S. K. (1993). On copy avoidance in single assignment languages. In D. S. Warren (Ed.), *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, pp. 393–407. The MIT Press.

Demoen, B., M. García de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey (1999). An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, Volume 1713 of *Lecture Notes in Computer Science*, pp. 174–188. Springer-Verlag.

Dowd, T., F. Henderson, and P. Ross (2001). Compiling Mercury to the .NET common language runtime. In N. Benton and A. Kennedy (Eds.), *Electronic Notes in Theoretical Computer Science*, Volume 59. Elsevier.

Dowd, T., Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery (1999). Run time type information in Mercury. In *Principles and Practice of Declarative Programming*, Volume 1702 of *Lecture Notes in Computer Science*, pp. 224–243. Springer-Verlag.

Falaschi, M., M. Gabbrielli, and K. Marriott (1993). Compositional analysis for concurrent constraint programming. In *Proceedings of the Eigth Annual IEEE Symposium on Logic in Computer Science (LICS)*, Los Alamitor, California, pp. 210–221. IEEE Computer Society Press.

Gabbrielli, M., G. Levi, and M. C. Meo (1996). Resultants semantics for Prolog. *Journal of Logic and Computation 6*(4), 491–521.

Gallagher, J. (2004). A program transformation for backwards analysis of logic programs. See Bruynooghe and Lau (2004). To appear.

García de la Banda, M., K. Marriott, P. Stuckey, and H. Søndergaard (1998, April). Differential methods in logic program analysis. *Journal of Logic Programming 35*(1), 1–37.

Genaim, S. and M. Codish (2001, December). Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov (Eds.), *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Volume 2250 of *Lecture Notes in Artificial Intelligence*, pp. 681–690. Springer-Verlag.

Giacobazzi, R., F. Ranzato, and F. Scozzari. Making abstract domains condensing. *ACM Transactions on Computational Logic*. To appear.

Giacobazzi, R. and F. Scozzari (1998). A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems 20*(5), 1067–1109.

Gordon, M. J. (1979). *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag.

Gosling, J. and H. McGilton (1995, October). The Java language environment – A whitepaper. Technical report, Sun Microsystems.

Gudjonsson, G. and W. Winsborough (1993). Update in place: Overview of the Siva project. In D. Miller (Ed.), *Proceedings of the International Logic Programming Symposium*, Vancouver, Canada, pp. 94–113. The MIT Press.

Hall, C. V., K. Hammond, S. L. P. Jones, and P. L. Wadler (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*(2), 109–138.

Hamilton, G. W. (1993). *Compile-Time Optimisation of Store Usage in Lazy Funtional Programs*. Ph. D. thesis, University of Stirling.

Hamilton, G. W. (1995, September). Compile-time garbage collection for lazy functional languages. In H. Baker (Ed.), *Proceedings of International Workshop on Memory Management*, Volume 986 of *Lecture Notes in Computer Science*, Department of Computer Science, Keele University. Springer-Verlag.

Heaton, A., M. Abo-Zaed, M. Codish, and A. King (2000). Simple, efficient and scalable groundness analysis of logic programs. *Journal of Logic Programming 45*(1-3), 143–156.

Henderson, F., T. Conway, Z. Somogyi, and D. Jeffery (1996, February). The Mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne.

Henderson, F. and Z. Somogyi (2002). Compiling Mercury to high-level C code. In R. N. Horspool (Ed.), *Compiler Construction : 11th International Conference, CC 2002*, Volume 2304 of *Lecture Notes in Computer Science*, Grenoble, France, pp. 197–212. Springer-Verlag.

Henglein, F. and C. Mossin (1994, April). Polymorphic binding-time analysis. In D. Sannella (Ed.), *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming*, Volume 788 of *Lecture Notes in Computer Science*, Edinburgh, U.K., pp. 287–301. Springer-Verlag.

Hill, P. M., R. Bagnara, and E. Zaffanella (1998). The correctness of set-sharing. In G. Levi (Ed.), *Static Analysis: Proceedings of the 5th International Symposium*, Volume 1503 of *Lecture Notes in Computer Science*, Pisa, Italy, pp. 99–114. Springer-Verlag.

Hill, P. M. and F. Spoto (2002). A foundation of escape analysis. In H. Kirchner and C. Ringeissen (Eds.), *Algebraic Methodology and Software Technology; Proceedings of 9th International Conference, AMAST 2002*, Volume 2422 of *Lecture Notes in Computer Science*, Reunion Island, France, pp. 380–395. Springer-Verlag.

Hindley, J. R. (1969, December). The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, 29–60.

Howe, J. M. and A. King (2000, March). Implementing groundness analysis with definite boolean functions. In G. Smolka (Ed.), *European Symposium on Programming*, Volume 1782 of *Lecture Notes in Computer Science*, pp. 200–214. Springer-Verlag.

Howe, J. M., A. King, and L. Lu (2004, May). Analysing logic programs by reasoning backwards. In M. Bruynooghe and K.-K. Lau (Eds.), *Program Development in Computational Logic*, Lecture Notes in Computer Science. Springer-Verlag.

Hudak, P., S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson (1992). Report on the programming language Haskell: a non-strict, purely functional language – version 1.2. *ACM SIGPLAN Notices 27*(5), 1–164.

Hughes, S. (1992). Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation 2*(4), 483–509.

Jacobs, D. and A. Langen (1992, May/July). Static analysis of logic programs for independent AND-parallelism. *Journal of Logic Programming 13*(2 &3), 291–314.

Jaffar, J. and M. J. Maher (1994, May/July). Constraint Logic Programming: A survey. *Journal of Logic Programming 19 & 20*, 503–581.

Janssens, G., M. Bruynooghe, and V. Dumortier (1995). A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In J. LLoyd (Ed.), *Logic Programming, Proceedings of the 1995 International Symposium (ILPS'95)*, Portland, Oregon, pp. 336–350. MIT Press.

Janssens, G., M. Hermenegildo, F. Bueno, M. García de la Banda, and A. Mulkers (1992, February). A review of some abstract interpretation systems. Report CW143, Department of Computer Science, Katholieke Universiteit Leuven.

Jeffery, D. (2002, February). *Expressive Type Systems for Logic Programming Languages*. Ph. D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

Jeffery, D., F. Henderson, and Z. Somogyi (1998, September). Type classes in Mercury. Technical Report 98/13, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia.

Jones, N. D. and H. Søndergaard (1987). A semantic-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin (Eds.), *Abstract Interpretation of Declarative Languages*, Ellis Horwood Series in Computers and their Applications, pp. 123–142. Chichester: Ellis Horwood.

Kågedal, A. (1995). *Exploiting Groundness in Logic Programs*. Ph. D. thesis, Linköping University, Dept. of Computer and Information Science, S-581 83 Linköping, Sweden. Linköping Studies in Science and Technology. No. 383.

Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language* (1st ed.). Prentice Hall.

King, A. (1994). A synergistic analysis for sharing and groundness which traces linearity. In D. Sannella (Ed.), *Proceedings of Fifth European Symposium on Programming Languages and Systems – ESOP'94*, Edinburgh, pp. 363–378. Springer-Verlag.

King, A. and L. Lu (2002, July). A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming 2*(4-5), 517–547.

Kluźniak, F. (1987). Type synthesis for ground Prolog. In J.-L. Lassez (Ed.), *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, pp. 788–816. MIT Press.

Lagoon, V., F. Mesnard, and P. J. Stuckey (2003, December). Termination analysis with types is more accurate. In C. Palamidessi (Ed.), *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, Volume 2916 of *Lecture Notes in Computer Science*, pp. 254–268. Springer-Verlag.

Lagoon, V. and P. J. Stuckey (2001, March). A framework for analysis of typed logic programs. In H. Kuchen and K. Ueda (Eds.), *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, Volume 2024 of *Lecture Notes in Computer Science*, pp. 296–310. Springer-Verlag.

Le Charlier, B. and P. Van Hentenryck (1993). Groundness analysis for Prolog: Implementation and evaluation of the domain Prop. In D. Schmidt (Ed.), *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPMA'93*, Copenhagen,Denmark, pp. 99–110. ACM Press.

Le Charlier, B. and P. Van Hentenryck (1994). Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Trans. Program. Lang. Syst. 16*(1), 35–101.

Leuschel, M., B. Martens, and D. De Schreye (1998). Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems 20*(1), 208 – 258.

Lindgren, T., J. Bevemyr, and H. Millroth (1995, June). Compiler optimizations in Reform Prolog: Experiments on the KSR-1 multiprocessor. Technical Report 110, Computing Science Department, Uppsala University.

Lloyd, J. W. (1987). *Foundations of Logic Programming*. Springer-Verlag.

Lu, L. and A. King (2002, September). Backward type inference generalises type checking. In M. Hermenegildo and G. Puebla (Eds.), *Proceedings of The 9th International Static Analysis Symposium (SAS 2002)*, Volume 2477 of *Lecture Notes in Computer Science*, pp. 85–101. Springer-Verlag.

Lu, L. and A. King (2004, April). Backward pair sharing analysis. In Y. Kameyama and P. Stuckey (Eds.), *Seventh International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag.

Makholm, H. (2000, 15–16 October). A region-based memory manager for Prolog. In *International Symposium on Memory Management*, Minneapolis, MN, USA. ACM.

Marriott, K. and H. Søndergaard (1993). Precise and efficient groundness analysis for logic programs. *ACM Lett. Prog. Lang. Syst. 2*(1-4), 181–196.

Marriott, K., H. Søndergaard, and N. D. Jones (1994, May). Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems 16*(3), 607–648.

Martelli, A. and U. Montanari (1982). An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*(2), 258–282.

Mazur, N., G. Janssens, and M. Bruynooghe (1999a, June). Towards memory reuse for Mercury. Report CW 278, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

Mazur, N., G. Janssens, and M. Bruynooghe (1999b). Towards modular liveness analysis for Mercury. In K. Sagonas and P. Tarau (Eds.), *Proceedings of the International Workshop on Implementation of Declarative Languages*, Paris, France, pp. 1–17.

Mazur, N., G. Janssens, and M. Bruynooghe (1999c). Towards modular liveness analysis for Mercury. In S. Etalle (Ed.), *Proceedings of the 1999 Benelux Workshop on Logic Programming*, pp. 1–17.

Mazur, N., G. Janssens, and M. Bruynooghe (2000). A module based analysis for memory reuse in Mercury. In J. L. et al (Ed.), *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, Volume 1861 of *Lecture Notes in Artificial Intelligence*, pp. 1255–1269. Springer-Verlag.

Mazur, N., P. Ross, G. Janssens, and M. Bruynooghe (2001). Practical aspects for a working compile time garbage collection system for Mercury. In P. Codognet (Ed.), *Proceedings of ICLP 2001 - Seventeenth International Conference on Logic Programming*, Volume 2237 of *Lecture Notes in Computer Science*. Springer-Verlag.

Microsoft. Microsoft .NET. http://www.microsoft.com/net/.

Milner, R. (1978). A theory of type polymorphism in programming languages. *Journal of Computer and System Science 17*(3), 348–375.

Milner, R., M. Tofte, and D. Macqueen (1997). *The Definition of Standard ML*. MIT Press.

Mohnen, M. (1995, May). Efficient compile-time garbage collection for arbitrary data structures. Technical Report 95–08, University of Aachen. Also in Seventh International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP95.

Mohnen, M. (1997). *Optimising the Memory Management of Higher-Order Functional Programs*. Ph. D. thesis, RWTH Aachen.

Morrisett, G. and J. Reppy (2000). The third annual ICFP programming contest. In Conjunction with the 2000 International Conference on Functional Programming, http://www.cs.cornell.edu/icfp/.

Mulkers, A. (1991, December). *Deriving Live Data Structures in Logic Programs by Means of Abstract Interpretation*. Ph.D. thesis, Department of Computer Science, Katholieke Universiteit Leuven.

Muthukumar, K. and M. Hermenegildo (1989). Determination of variable dependence information through abstract interpretation. In E. Lusk and R. Overbeek (Eds.), *Proceedings of the North American Conference on Logic Programming*, Cambridge, pp. 166–185. MIT Press.

Muthukumar, K. and M. Hermenegildo (1991). Combined determination of sharing and freeness of program variables through abstract interpretation. In K. Furukawa (Ed.), *Proceedings of the Eighth International Conference on Logic Programming*, Paris, pp. 49–63. MIT Press, Cambridge.

Muthukumar, K. and M. Hermenegildo (1992, July). Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13*(2&3), 315–347.

Mycroft, A. and R. A. O'Keefe (1984). A polymorphic type system for Prolog. *Artificial Intelligence 23*(3), 295–307.

Nethercote, N. (2001, September). The analysis framework of HAL. Master's thesis, Department of Computer Science and Software Engineering, University of Melbourne, Australia.

Nielson, F. (1982). A denotational framework for data flow analysis. *Acta Informatica 18*, 265–287.

Nielson, F. (1988). Strictness analysis and denotational abstract interpretation. *Information and Computation 76*(1), 29–92.

Nielson, F., H. R. Nielson, and C. Hankin (1999). *Principles of Program Analysis*. Springer-Verlag.

Nielson, H. R. and F. Nielson (1992). *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing. Chichester: John Wiley & Sons, Inc.

Nielson, H. R. and F. Nielson (1996). Semantics with applications: Model-based program analysis. Technical Report DAIMI FN-61, Aarhus University, Denmark.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Pfenning, F. (1992). *Types in Logic Programming*. MIT Press.

Platt, D. S. (2003). *Introducing Microsoft@ .NET, Third Edition*. Microsoft Press.

Puebla, G., J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey (2004). A generic framework for context-sensitive analysis of modular programs. See Bruynooghe and Lau (2004). To appear.

Puebla, G. and M. Hermenegildo (1999a, November). Abstract multiple specialization and its application to program parallelization. *Journal of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41*(2&3), 279–316.

Puebla, G. and M. Hermenegildo (1999b). Some issues in analysis and specialization of modular Ciao-Prolog programs. In M. Leuschel (Ed.), *Proceedings of the Workshop on Optimization and Implementation of Declarative Languages*, Las Cruces. In Electronic Notes in Theoretical Computer Science, Volume 30 Issue No.2, Elsevier Science.

Shaham, R., E. K. Kolodner, and M. Sagiv (2001). Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pp. 104–113. ACM Press.

Shaham, R., E. Yahav, E. Kolodner, and M. Sagiv (2003, June). Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of Static Analysis Symposium (SAS'03)*, Volume 2694 of *Lecture Notes in Computer Science*, pp. 483–503. Springer-Verlag.

Somogyi, Z., F. Henderson, and T. Conway (1996, October-December). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29*(1–3), 17–64.

Speirs, C., Z. Somogyi, and H. Søndergaard (1997). Termination analysis for Mercury. In *Proceedings of the 4th International Symposium on Static Analysis*, Volume 1302 of *Lecture Notes in Computer Science*, pp. 160–171. Springer-Verlag.

Steele, G. L. (1984). *COMMON LISP: The language*. Bedford, USA: Digital Press. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.

Sterling, L. and E. Shapiro (1986). *The Art of Prolog*. MIT Press.

Stuckey, P. J. and K. Marriott (1998). *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Mass.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math. 5*, 285–309.

The Mercury Team (2000). ICFP 2000: The Merry Mercurians. Description of the Mercury entry to the ICFP'2000 programming contest, http://www.mercury.cs.mu.oz.au/information/events/icfp2000.html.

Tofte, M. and L. Birkedal (1998, July). A region inference algorithm. *ACM Transactions on Programming Languages and Systems 20*(4), 734–767.

Tofte, M., L. Birkedal, M. Elsman, N. Hallenberg, T. H. O. Højfeld, P. Sestoft, and P. Bertelsen (1997). Programming with regions in the ML Kit. Technical Report D-342, Dept. of Computer Science, University of Copenhagen.

Tofte, M. and J.-P. Talpin (1997). Region-based memory management. *Information and Computation 132*(2), 109–176.

Van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. MIT Press.

Van Hentenryck, P., A. Cortesi, and B. Le Charlier (1995). Type analysis of Prolog using type graphs. *Journal of Logic Programming 22*(3), 179–209.

Vandecasteele, H. (1999a, may). *Constraint Logic Programming: Applications and Implementation*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium. 226+vii pagesURL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=21836.

Vandecasteele, H. (1999b). MROPE II: A finite domain solver on top of Mercury. In *Proceedings of the 1999 ERCIM/COMPULOG Workshop on Constraints*, pp. 1–13. URL = http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=18802.

Vandecasteele, H., B. Demoen, and G. Janssens (2000). A finite domain CLP solver on top of Mercury. In *New Trends in Constraints : Joint ERCIM/Compulog Workhsop, Paphos, Cyprus, October 1999, Selected Papers*, Volume 1865 of *Lecture Notes in Artificial Intelligence*, pp. 256–273. ERCIM/Compulog: Springer-Verlag.

Vanhoof, W. (2001, June). *Techniques for On- and Off-line Specialisation of Logic Programs*. Ph. D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium. Pages: xiv+323+xxxiii.

Vanhoof, W. and M. Bruynooghe (1999). Binding-time analysis for Mercury. In D. De Schreye (Ed.), *Proceedings of the 16th International Conference on Logic Programming*, pp. 500–514. MIT Press.

W3C. Hypertext markup language (HTML). http://www.w3.org/MarkUp.

Wadler, P. (1992, January). The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albequerque, New Mexico, pp. 1–14.

Wadler, P. and S. Blott (1989, January). How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 60–76. ACM.

Wilson, P. R. (1992, 16–18 September). Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen (Eds.), *Proceedings of International Workshop on Memory Management*, Volume 637 of *Lecture Notes in Computer Science*, St Malo, France. Springer-Verlag.

Winsborough, W. (1992, May/July). Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming 13*(2 &3), 259–290.

Zaffanella, E., R. Bagnara, and P. M. Hill (1999). Widening Sharing. In G. Nadathur (Ed.), *Principles and Practice of Declarative Programming*, Volume 1702 of *Lecture Notes in Computer Science*, Paris, France, pp. 414–431. Springer-Verlag.

Biography

Nancy Mazur was born on the 19th of December 1974 in Leuven. She finished High School at the *Lycée Emile Jacqmain* in Brussels in 1992. In 1994 she received a Bachelor's degree of Science in Engineering (*Kandidaat Burgerlijk Ingenieur*) and in 1997 a Master's degree of Science in Engineering in Computer Science (*Burgerlijk Ingenieur in de Computerwetenschappen*) from the Katholieke Universiteit Leuven in Belgium. Her master thesis with the title "Declarative I/O within a Functional Logic Language: Determinism Analysis" was supervised by Professor Danny De Schreye and Professor Michael Hanus (RWTH Aachen). In 1996 she participated with the European student exchange project ERASMUS and spent five months studying at the RWTH Aachen.

In September 1997 she started working at SIEMENS ATEA, in Herentals. In February 1998 she came back to her Alma Mater where she joined the DTAI research group and started working as a Ph.D. student, funded by a K.U.Leuven Research grant. She was a visiting scholar at the University of Melbourne, Australia, working with the Mercury Team under the guidance of Professor Zoltan Somogyi from January to February 2000.

Hergebruik van afgedankt geheugen tijdens de vertaling in de declaratieve programmeertaal Mercury

Nancy MAZUR

Beknopte Samenvatting

Een belangrijk aspect van hoog niveau programeertalen is het voorzien in *automatisch geheugenbeheer* waarbij de programmeur verlost wordt van de foutgevoelige taak van expliciet geheugen te reserveren en na gebruik weer vrij te geven. Een gekende manier van automatisch geheugenbeheer is gebruik te maken van een *garbage collector*. Conceptueel is dit een afzonderlijk programma dat samen met het gebruikersprogramma wordt uitgevoerd en dat instaat voor het toekennen van geheugen aan dat programma, alsook voor het vrijgeven van dat geheugen van zodra het door het gebruikersprogramma niet meer gebruikt wordt.

Terwijl bovengenoemde garbage collector voor automatisch geheugenbeheer zorgt tijdens de *uitvoering* van het programma, zo bestaat er een complementaire vorm van automatisch geheugenbeheer die het geheugengebruik van een programma onderzoekt en optimaliseert tijdens de *vertaling* van dat programma. De vertaler onderzoekt de levensduur van de variabelen en voegt de nodige instructies toe voor het al dan niet vrijgeven of herbruiken van het geheugen dat met deze variabelen overeenstemt.

In deze thesis bestuderen we de tweede vorm van automatisch geheugenbeheer in de context van de declaratieve taal Mercury. Declaratieve talen hebben de eigenschap dat ze elke vorm van door de programmeur gecontroleerd expliciet herbruik van geheugen verbieden. Een gevolg hiervan is dat programma's in deze talen een hoog geheugenverbruik vertonen, wat het belang van automatisch geheugenbeheer in het algemeen, en geheugenbeheer tijdens de vertaling in het bijzonder, voor deze talen benadrukt. Bovendien vergemakkelijkt de wiskundige achtergrond van Mercury de realisatie van de programma analyses waar het uiteindelijke geheugenbeheer tijdens de vertaling op steunt.

Hiervoor definiëren we een aantal wiskundig onderlegde semantieken voor de programmeertaal Mercury en tonen formeel hun equivalentie aan. We gebruiken deze semantieken om de programma analyses die deel uit maken van het geheugenbeheer systeem uit te werken. We breiden het geheugenbeheer systeem uit zodat het op een efficiënte wijze met een modulaire structuur van programma's kan omgaan. Tenslotte implementeren we dit geheugenbeheer systeem in de Melbourne Mercury vertaler. Voor zover we weten is dit de meest volledige implementatie van dit type van geheugenbeheer voor een in de praktijk gebruikte programmeertaal.

1 Inleiding

Het doel van deze thesis is het ontwikkelen van een automatisch geheugenbeheer systeem dat tijdens de vertaling van een gegeven programma bepaalt hoe het geheugen best gebruikt wordt. Gezien het hoge geheugengebruik bij declaratieve programma's enerzijds en hun wiskundige achtergrond anderzijds verwezenlijken we dit systeem in de specifieke context van de declaratie taal Mercury.

Een van de belangrijkste aspecten van hoog niveau programmeertalen is dat ze trachten zo veel mogelijk technische laag niveau details rond het programmeren te verbergen opdat de programmeur zich uitsluitend op het eigenlijke software probleem kan concentreren. Het beheren van het geheugen is een moeilijke en foutgevoelige taak, en zo is het dus niet verwonderlijk dat men net deze taak uit de handen van de programmeur wil nemen. De meest gangbare wijze om dit te realiseren is door gebruik te maken van een *geheugenrecuperatie* programma (Eng. *garbage collector*). Conceptueel gezien is dit een afzonderlijk programma dat samen met het gebruikersprogramma wordt uitgevoerd en dat instaat voor het toekennen van geheugen aan dat programma, alsook voor het vrijgeven van dat geheugen van zodra het door het gebruikersprogramma niet meer gebruikt wordt. Het vrijgeven gebeurt door regelmatig het geheugen van het gebruikersprogramma te onderzoeken en na te gaan welke geheugencellen niet meer bereikt kunnen worden vanuit dat programma. Dit zijn de zogenoemde *afgedankte geheugencellen*, in tegenstelling tot de *actieve geheugencellen* die wel nog door het programma zouden kunnen gebruikt worden.

Het gebruik van een geheugenrecuperatie programma bevrijdt weliswaar de programmeur van het zelf beheren van dat geheugen, maar kan ook een aantal nadelen hebben. Zo neemt het geheugenrecuperatie programma zelf geheugen en uitvoeringstijd in beslag. Met name voor tijdskritische toepassingen is deze kost niet altijd aanvaardbaar. Tijdens de uitvoering van een programma heeft men vaak ook niet alle gegevens ter beschikking. Dit betekent dat het geheugenrecuperatie programma een overschatting zal moeten maken van de actieve geheugencellen. Zodoende verkrijgt men een groter geheugenverbruik van het gebruikersprogramma dan eventueel verwacht. Tenslotte worden afgedankte geheugencellen slechts gerecupereerd wanneer het recuperatieprogramma actief wordt. Het kan dus best zijn dat afgedankte geheugencellen een hele poos onnodig in het geheugen bewaard worden, wat dan weer een negatieve invloed heeft op het algemeen geheugengebruik van het gebruikersprogramma.

Deze gebreken kunnen gedeeltelijk verholpen worden door tijdens het vertalen van het gebruikersprogramma het geheugengebruik van dat programma te analyseren, en zodoende te optimaliseren. Dit leidt ons tot een nieuwe vorm van automatisch geheugenbeheer: *geheugenherbruik tijdens de vertaling*. Hierbij onderzoekt de vertaler de levensduur van de variabelen en voegt dan de nodige instructies toe voor het al dan niet vrijgeven of herbruiken van het geheugen

dat met deze variabelen overeenstemt. Het resultaat is een vertaald programma waarin het vrijgeven en herbruik van het geheugen gedeeltelijk in het programma gecodeerd is. Men kan dan verwachten dat hierdoor het gebruik van een afzonderlijk geheugenrecuperatieprogramma aanzienlijk kan verminderd worden en het algemeen geheugengebruik van het gebruikersprogramma verbeterd wordt.

Zoals reeds aangekondigd ontwikkelen we in deze thesis een geheugenherbruik systeem tijdens de vertaling voor de specifieke context van de declaratieve taal Mercury. Declaratieve talen hebben de eigenschap dat echt elke vorm van expliciet manueel geheugenbeheer onmogelijk wordt gemaakt. Zo kan de programmeur de waarde van een variabele niet meer wijzigen eens zij of hij er een waarde aan heeft toegekend. Indien zij/hij een andere waarde wenst te gebruiken moet zij/hij daartoe een andere variabele aanmaken. Dit principe is essentieel voor declaratieve talen, maar heeft natuurlijk als gevolg dat het geheugengebruik van programma's in deze talen aanzienlijk groot is. Een goed automatisch geheugenbeheer systeem is daarom onontbeerlijk en we verwachten dan ook dat het gebruik van een geheugenherbruik systeem tijdens de vertaling voor deze talen merkbare resultaten geeft. Bovendien vergemakkelijkt de wiskundige achtergrond van declaratieve talen in het algemeen, en Mercury in het bijzonder, de realisatie van deze vorm van geheugenbeheer.

De belangrijkste bijdragen van dit werk zijn als volgt: we definiëren een aantal wiskundig onderlegde semantieken voor de programmeertaal Mercury en tonen formeel hun equivalentie aan. We gebruiken deze semantieken om de programma analyses die deel uit maken van het geheugenbeheer systeem uit te werken. We breiden het geheugenbeheer systeem uit zodat het op een efficiënte wijze met een modulaire structuur van programma's kan omgaan. Tenslotte implementeren we dit geheugenbeheer systeem in de Melbourne Mercury vertaler.

Het vervolg van deze samenvatting is als volgt opgebouwd. In paragraaf 2 stellen we de taken van een geheugenherbruik systeem tijdens de vertaling voor aan de hand van een intuïtief voorbeeld. We beschrijven de programmeertaal Mercury in paragraaf 3 en definiëren de semantiek van programma's in deze taal in paragraaf 4. Op basis van de notie van een *geheugenstructuur* (paragraaf 5) definiëren we de programma analyses die bij deze vorm van geheugenbeheer nodig zijn: het bepalen van de *gedeelde geheugenstructuren*, het bepalen van de *actieve geheugenstructuren*, en tenslotte, het afleiden van het *herbruik van geheugenstructuren*. Dit vormt de inhoud van paragraaf 6. In paragraaf 7 passen we deze analyses aan aan de modulaire structuur van Mercury programma's. In paragraaf 8 beschrijven we enkele praktische problemene die nog aangepast moesten worden om tot een performante implementatie van het voorgestelde geheugenbeheer systeem te komen. Ook de resultaten worden in die paragraaf kort besproken. In de thesis ontwikkelden we ook een raamwerk voor het afleiden van optimalisaties waarover we in paragraaf 9 berichten. In paragraaf 10 halen we de voornaamste onderzoeksdomeinen aan die met deze thesis verband houden waarop we dan in

paragraaf 11 deze samenvatting besluiten.

2 Intuïtief Voorbeeld

Laat ons het eenvoudige voorbeeld nemen van een gegevensbank waarin men de personeelsgegevens van een bedrijf wenst bij te houden. Indien we veronderstellen dat een werknemer eenvoudigweg beschreven wordt door zijn naam, geboortedatum en loon, dan kan men dat in Mercury voorstellen door een *term* met drie *argumenten*. Het predikaat om het loon van een werknemer aan te passen zou er dan als volgt uit kunnen zien.

Voorbeeld 1

```
updateSalary(EmployeeRecord, NewSalary, NewRecord) :-
    EmployeeRecord = employee(Name, Birthday, OldSalary),
    NewRecord = employee(Name, Birthday, NewSalary).
```

In deze definitie wordt de oorspronkelijke term die de gegevens van een bepaalde werknemer voorstelt niet gewijzigd, maar gaat men een nieuwe term aanmaken waarin men de naam en de geboortedatum van de oorspronkelijke term overneemt, en waarbij nu het loon gelijk wordt gesteld aan het nieuwe loon.

Stel dat dit predikaat deel uit maakt van een grotere administratieve toepassing en stel dat tijdens de uitvoering van deze toepassing een variabele gebruikt wordt JackRecord die op dat moment gebonden is aan de term:

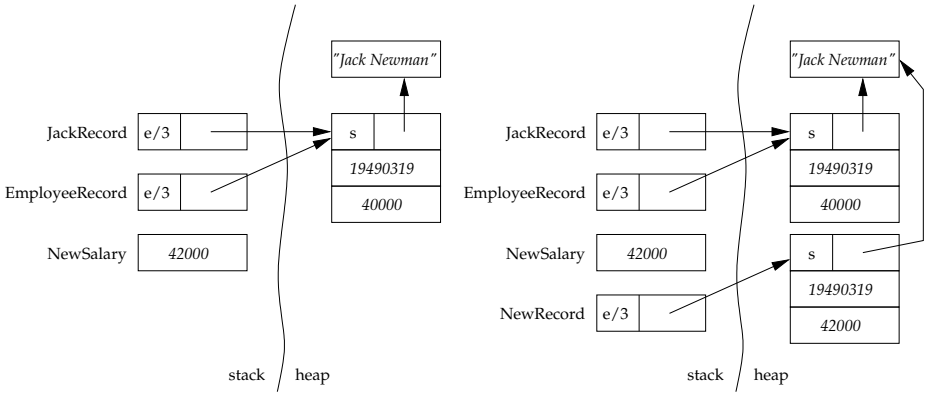
```
employee("Jack␣Newman", 19490319, 40000)
```

Deze term stelt dan de gegevens voor van de werkgever Jack Newman, geboren op 19 maart 1949, en die momenteel 40000EUR per jaar verdient. Men kan het loon van deze werkgever wijzigen door bovenstaand predikaat op te roepen, bijvoorbeeld als volgt:

```
updateSalary(JackRecord, 42000, NewJackRecord)
```

Figuur 1 geeft schematisch de toestand van het geheugen weer *vóór* (a) en *na* (b) het uitvoeren van deze predikaat-oproep.

Indien tijdens de uitvoering van dit predikaat blijkt dat de term waaraan variabele JackRecord gebonden is nergens elders meer door het programma gebruikt zal worden, dan kan men na de uitvoering van dat predikaat de geheugencellen die deze term voorstellen als afgedankte geheugencellen bestempelen (Figuur 2 (a)). Bij gebruik van een geheugenrecuperatie programma zullen deze afgedankte geheugencellen vrijgegeven worden tijdens één van de eerstvolgende interventies van dat programma. Ondertussen zal men echter nieuw geheugen hebben moeten toekennen om de term gebonden aan NewJackRecord weer te geven. Het is duidelijk dat men hier het geheugengebruik zou willen optimaliseren door de



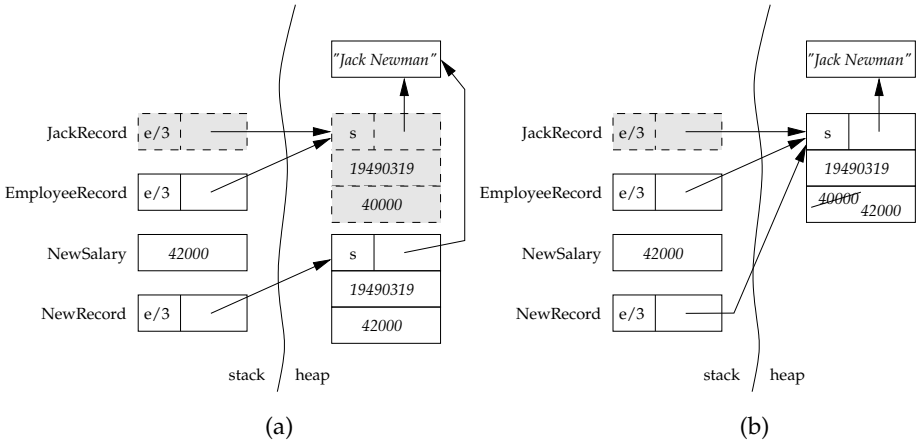
Figuur 1: Toestand van de geheugencellen vóór (a) en na (b) de oproep updateSalary(JackRecord, 42000, NewJackRecord), waar JackRecord gebonden is aan de term employee("Jack␣Newman", 19490319, 40000).

afgedankte geheugencellen rechtstreeks te herbruiken voor het voorstellen van de nieuwe term (Figuur 2 (b)). Dit kan inderdaad gerealiseerd worden indien men tijdens de vertaling van het predikaat kan achterhalen dat bij sommige oproepen van dat predikaat de term waaraan het eerste argument gebonden is, na de oproep altijd afgedankte geheugencellen oplevert.

Deze vorm van geheugenherbruik is het doel van het geheugenherbruik tijdens vertaling die we in deze thesis beogen. Uit dit voorbeeld wordt duidelijk wat er allemaal bij zo een systeem komt kijken.

- Tijdens de vertaling van het programma moet het geheugenherbruik systeem achterhalen wanneer welke geheugencellen die tijdens de uitvoering aangemaakt worden afgedankt worden. Deze informatie kan men verkrijgen door middel van programma analyse, en meer specifiek *abstract interpretatie*. Deze techniek laat toe eigenschappen over de uitvoering van een programma af te leiden, zonder het programma evenwel uit te voeren. De centrale eigenschap bij dit geheugenherbruik systeem is de kennis over de afgedankte cellen, of diens duale vorm, de kennis over de actieve geheugencellen.

- Indien men te weten komt welke cellen actief zijn, en zodus welke cellen afgedankt zijn, zo is het de taak van het geheugenherbruik systeem om te beslissen wat met de afgedankte cellen gedaan moet worden. Gaat men dat geheugen vrijgeven, of gaat men het rechtstreeks herbruiken?



Figuur 2: (a) Afgedankte geheugencellen (grijs). (b) Rechtstreeks herbruik van de afgedankte geheugencellen.

- Natuurlijk is het niet voldoende om bovenstaande informatie gewoon af te leiden, maar moet men deze informatie ook gebruiken wanneer men de vertaalde code van het geanalyseerde programma gaat genereren. Omdat waarschijnlijk niet bij elke oproep elke vorm van gededecteerd geheugenherbruik toegelaten is zal men éénzelfde predikaat vaak ontkoppelen in meerdere versies die dan elk een voor het programma interessante vorm van geheugenherbruik realiseren.
- Uit het updateSalary voorbeeld is duidelijk dat het herbruik van de geheugencellen van de betrokken term enkel en alleen toegestaan mag worden indien het programma deze geheugencellen daadwerkelijk niet meer gebruikt. Het herbruik zal dus afhangen van de geheugentoestand van het programma op het moment dat het predikaat opgeroepen wordt. Deze informatie kan men bij programma's die slechts uit één enkele module bestaan prima achterhalen. In de aanwezigheid van modules echter wordt het ingewikkelder: elke module wordt afzonderlijk vertaald, zodus kan men niet meteen weten hoe een predikaat gedefinieerd in een module A gebruikt zal worden vanuit een module B. Desalniettemin willen we dat er geheugenherbruik plaats vindt voor zover de veiligheid van het programma gewaarborgd kan blijven. Hierbij stellen zich twee specifieke vragen:
  - Indien men een predikaat analyseert waaruit blijkt dat er geheugenherbruik mogelijk is, is het dan de moeite om een variant van dit predikaat aan te maken die dat geheugenherbruik inderdaad verwezenlijkt? Zal deze variant ooit tijdens de uitvoering van het programma

gebruikt worden, of blijft het gewoon dode code die onnodig het programma vergroot?
- Stel dat we inderdaad meerdere varianten voorzien voor een bepaald predikaat, dan willen we weten wanneer deze varianten mogen opgeroepen worden zonder de veiligheid van het uiteindelijke programma in gevaar te brengen? Hoe gaan we de voorwaarden opstellen die deze veiligheid garanderen?

Uit bovenstaande is duidelijk dat een geheugenherbruik systeem tijdens de vertaling een niet-triviaal probleem is en dat er heel wat programma analyse bij te pas komt. Opdat het systeem echter bruikbaar zou zijn in echte programmeeromgevingen moeten we er ook voor zorgen dat de kost van het gebruik van dit systeem opweegt tegenover de winst in geheugengebruik die men er mee kan verkrijgen. Dit betekent dat de onderliggende analyses voldoende snel moeten zijn zonder hierbij aan precisie van de resultaten te verliezen. Dit op zich vormt een uitdaging.

3 Mercury

Mercury is een pure logische programmeertaal met enkele kenmerken van een functionele taal. Het werd ontwikkeld aan de University of Melbourne in Australië (Somogyi, Henderson, and Conway 1996). Bij de specificatie van deze taal heeft men nauwlettend geprobeerd om de gekende problemen van de meest gangbare logische programmeertalen te vermijden, we denken hierbij vooral aan de taal Prolog (Sterling and Shapiro 1986). Het resultaat is een *pure* taal, met expliciete ondersteuning voor het ontwikkelen van grote toepassingen door een team van programmeurs, waarbij de vertaler sterke foutmeldingen tracht te geven. Tenslotte moeten Mercury programma's resulteren in snelle en efficiënte code.

In deze thesis beperken we ons tot eerste-orde logische programma's. Het alfabet voor deze programma's bestaat uit een eindige verzameling variabelen, $\mathcal{V}$, een eindige verzameling functie-symbolen Σ alsook een eindige verzameling predikaat symbolen Π. Met functie-symbolen en predikaat-symbolen associeert men doorgaans een *ariteit*. Dit is een natuurlijk getal dat het aantal *argumenten* van deze entiteiten weergeeft. Functie-symbolen met ariteit nul worden *constanten* genoemd. Een *term* is een variabele of een functie-symbool $f/n \in \Sigma$ toegepast op n argumenten. Deze argumenten zijn elk opnieuw termen. Een *atoom* is een predikaat-symbool $p/n \in \Pi$ toegepast op een sequentie van n termen. Het predicaat symbool $=/2$ wordt op speciale wijze behandeld, en noemen we *expliciete unificatie*. We beperken de betekenis van *atoom* daarom tot atomen die geen expliciete unificatie zijn. Een *literal* is dan een atoom, of een expliciete unificatie.

Een *expressie* is een term, een literal, of een compositie van deze elementen. Een *expressie* is *gegrond* (Eng. *ground*) indien ze geen variabelen bevat.

In Mercury is een *goal* een *conjunctie* van goals, een *disjunctie* van goals, een *voorwaardelijke uitdrukking* van goals, een *negatie* van een goal, of gewoon een literal. Een *predikaat* bestaat dan uit een *hoofd*, en een *lichaam*. Het hoofd wordt voorgesteld door een atoom, terwijl het lichaam een goal is. De volledige syntax wordt weergegeven in Figuur 3.

In Mercury programma's zoals ze door een programmeur moeten geschreven worden verwacht men dat de programmeur elk van zijn predikaten zorgvuldig annoteert met *type*, *mode* en *determinisme* informatie. De vertaler gebruikt type-informatie om aan elke variabele een type toe te kennen. Mode-informatie beschrijft de gegrondheid van elk van de variabelen bij het gebruik van een predikaat, terwijl determinisme informatie het aantal oplossingen van een predikaat weergeeft. In onze formele syntax wordt deze informatie niet expliciet overgenomen maar voorzien we eenvoudige operaties om, waar nodig, deze informatie eventueel op te vragen. Een deel van de mode-informatie is echter wel zichtbaar in de formele syntax. Mode-informatie laat de vertaler namelijk toe om elk van de expliciete unificaties te specialiseren naar één van de volgende vier vormen:

- *constructie*: hierbij wordt een nieuwe term opgebouwd en aan een variabele toegekend;
- *deconstructie*: de term waaraan een variabele is toegekend wordt ontleed om de individuele argumenten aan te kunnen spreken;
- *toekenning*: een variabele wordt gebonden aan de term waaraan een andere variabele gebonden is;
- *test*: de termen waaraan de twee betrokken variabelen gebonden zijn worden vergeleken.

In deze thesis spelen vooral de deconstructie en de constructie een rol. Het geheugenherbruik systeem zal bij deconstructies trachten te achterhalen of de ontleedde term niet toevallig tot afgedankte geheugencellen leidt, en zo ja, zal het deze afgedankte geheugencellen bij constructies laten herbruiken.

4 Mercury Semantiek

De techniek van *abstracte interpretatie* (Cousot and Cousot 1977; Cousot and Cousot 1992a; Bruynooghe 1991) laat toe om eigenschappen over een programma te weten te komen zonder deze programma's uit te voeren. Deze eigenschappen zijn waardevol voor een insectenverdelger, code optimalisatie, programma transformatie en andere soortgelijke toepassingen. Typische eigenschappen die men via abstracte interpretatie kan afleiden zijn: type informatie (Kluźniak 1987; Van Hentenryck, Cortesi, and Le Charlier 1995), groundness informatie (Kågedal 1995;

$$
\begin{array}{lcll}
\textit{Program} & ::= & r\,;\,q & \\
\textit{RuleBase} & ::= & \{p_1 \ldots p_{n_p}\} & n_p \geq 1 \\
\textit{Procedure} & ::= & p(\overline{X}) \leftarrow g & \\
\textit{Goal} & ::= & g_1, g_2 & \\
 & | & g_1; g_2 & \\
 & | & \text{if } g_1 \text{ then } g_2 \text{ else } g_3 & \\
 & | & \text{not } g & \\
 & | & l & \\
\textit{Literal} & ::= & X \Rightarrow f(\overline{Y}) & \text{(deconstructie)} \\
 & | & X \Leftarrow f(\overline{Y}) & \text{(constructien)} \\
 & | & X == Y & \text{(test)} \\
 & | & X := Y & \text{(toekenning)} \\
 & | & p(\overline{Y}) &
\end{array}
$$

waar

$$
\begin{array}{lcl}
r & \in & \textit{RuleBase} \\
\{p_1, \ldots, p_{n_p}\} & \subseteq & \textit{Procedure} \\
\{q, g, g_1, g_2, g_3\} & \subseteq & \textit{Goal} \\
l & \in & \textit{Literal} \\
\{X, Y, X_1, \ldots, X_n, Y_1, \ldots, Y_m\} & \subseteq & \mathcal{V} \\
f/n & \in & \Sigma
\end{array}
$$

Figuur 3: Formele syntax van eerste-orde Mercury programma's. Hierin stelt n_p het aantal predikaten voor in het programma, en $Y_1, \ldots, Y_m$ and $\overline{X}$ wordt gebruikt om een rij van afzonderlijk verschillende variabelen voor te stellen.

Cortesi, Filé, and Winsborough 1991; Marriott and Søndergaard 1993), gedeeldheid van vrije variabelen (Jacobs and Langen 1992; Muthukumar and Hermenegildo 1989), of combinaties van deze eigenschappen (King 1994; Cortesi and Filé 1991; Muthukumar and Hermenegildo 1991; Bagnara, Zaffanella, and Hill 2000).

Meestal wordt abstracte interpretatie geformuleerd in termen van de *operationele semantiek* van de taal. Men gaat hierbij een *concreet* domein definiëren — dit formaliseert de eigenschap die men van de uitvoering van het programma wenst te beschrijven, waarop men vervolgens de operaties definieert die het effect van de uitvoering van een programma in die taal weergeven. Met *vertaalt* vervolgens zowel het domein als de operaties naar een *abstract domein* en overeenstemmende *abstracte operaties*. Het resultaat is dan een beschrijving van de beoogde programma analyse. Om te vermijden dat men voor elke nieuwe analyse alle operaties opnieuw moet definiëren heeft men zogenoemde *raamwerken*

ontwikkeld, waaronder (Bruynooghe 1991) de meest bekende is voor de context van logisch programmeren. Zolang de onderliggende talen en analyses dezelfde operationele semantiek volgen, kan men een zelfde raamwerk gebruiken om al dan niet meerdere analyses te beschrijven.

In dit werk gaan we echter de *denotationele* semantiek van de taal gebruiken als basis voor het definiëren van de programma analyses. Het essentiële verschil is dat men hierbij een betekenis geeft aan programma's geschreven in die taal, los van de manier waarop de uitvoering van programma's in die taal, of analyses voor programma's voor die taal gespecifieerd zijn. Een programma wordt typisch uitgevoerd door een gegeven doelpredikaat op te lossen waarbij dan één voor één elk van de tegengekomen predikaat-oproepen afgehandeld worden. Indien een predikaat meerdere keren opgeroepen wordt, zo zal het programma elk van deze predikaat-oproepen afzonderlijk behandelen. Men spreekt van een *top/-down* implementatie van de taal. Bij programma analyse kan het echter zijn dat een top/down implementatie niet ideaal is en dat een *bottom/up* implementatie beter is. Dit is dan een andere operationele semantiek. Om dan de correctheid van de analyses aan te tonen moet men eerst aantonen dat de semantieken overeenstemmen. Dit werk kunnen we ons besparen door gebruik te maken van de denotationele aanpak. Dit is ook de reden waarom Marriott, Søndergaard, and Jones (1994) de notie van *denotationele abstracte interpretatie* hebben ingevoerd.

In deze aanpak gaat men een *meta-taal* gebruiken waarin men de denotationele semantiek van een programmeertaal uitdrukt. Deze semantiek bevat dan één belangrijke vrijheidsgraad, namelijk het exacte domein dat de te beschrijven eigenschappen weergeeft alsook de basisoperaties dat elementen van dit domein correct manipuleert. Men kan dan de semantiek *instantiëren* met een bepaald domein om het gedrag van het programma weer te geven m.b.t. die eigenschappen. Indien $\mathcal{S}em$ een gegeven semantiek is voor een taal, en A een domein en diens eigenschappen, dan gebruiken we de notatie $\mathcal{S}em(A)$ als het resultaat van het instantiëren van $\mathcal{S}em$ met dat domein. Hierbij moeten deze domeinen wel volledige tralies (Eng. *complete lattice*) zijn.

Marriott, Søndergaard, and Jones (1994) hebben dan aangetoond in hun centraal Theorema 4.4 dat voor een gegeven semantiek $\mathcal{S}em$ en twee volledige tralies A en B die met elkaar gerelateerd zijn volgens een *concretisatiefunctie* γ die aan een aantal voorwaarden voldoet die we hier achterwege laten, de eigenschappen beschreven door $\mathcal{S}em(A)$ gerelateerd zullen zijn aan de eigenschappen beschreven door $\mathcal{S}em(B)$ volgens dezelfde concretisatiefunctie γ, indien de operaties gedefinieerd over A ook volgens deze functie γ gerelateerd zijn aan de operaties gedefinieerd over B. Zodoende, indien A het *abstracte* domein voorstelt dat een correcte benadering weergeeft van de concrete eigenschappen die door het domein B worden voorgesteld, zo zal $\mathcal{S}em(A)$, de programma analyse, een correcte benadering geven van de concrete uitvoering van het programma, weergegeven door $\mathcal{S}em(B)$. De concretisatie-functie γ formaliseert hierbij de notie van "correc-

te benadering".

Dit theorema is daarom ook essentieel in deze thesis. Het laat ons namelijk toe om de correctheid van onze programma analyses aan te tonen door eenvoudigweg de correcte relatie op te stellen tussen een *concreet domein* en diens concrete operaties, en een *abstract domein* en de overeenstemmende abstracte operaties, onafhankelijk van de operationele semantiek van de programma analyses t.o.v. het eigenlijke concrete uitvoeringsmechanisme.

In Hoofdstuk 5 van de thesis gebruiken we (Marriott, Søndergaard, and Jones 1994) als basis om een aantal semantieken voor Mercury programma's op te stellen. We vertrekken hierbij van een *natuurlijke semantiek*, $\mathcal{S}em_M$ genoemd, om dan uit te komen bij een *oproep-onafhankelijk gebaseerde semantiek*, $\mathcal{S}em_{M\bullet}$.

De natuurlijke semantiek is inherent *oproep-afhankelijk*. Dat betekent dat men een predikaat een betekenis kan geven enkel en alleen in de context van een specifieke oproep naar dat predikaat. Dit komt ook overeen met de manier waarop men doorgaans een logisch programma gaat opvatten, wat onze benaming *natuurlijke semantiek* verklaart. Een oproep-onafhankelijke gebaseerde semantiek bestaat uit twee delen: een eerste deel definieert de betekenis van elk van de voorkomende predikaten binnen een programma *onafhankelijk* van de manier waarop deze predikaten zullen opgeroepen worden. In een tweede deel geeft men dan een oproep-gerichte betekenis aan het volledige programma door telkens gebruik te maken van de oproep-onafhankelijke betekenis van de predikaten. Het essentiële voordeel van deze semantiek is dat het ideaal is om gebruikt te worden bij analyses in de context van *modules*. Tijdens de analyse van een module weet men namelijk niet hoe de predikaten die er in gedefinieerd worden door het programma gebruikt zullen worden. Dit maakt een oproep-afhankelijke analyse onmogelijk. Bij een oproep-onafhankelijke aanpak kan men deze predikaten echter wel analyseren, om dan de bekomen analyse-resultaten verder te kunnen gebruiken. Een tweede belangrijk voordeel is dat men in het oproep-onafhankelijke deel van een oproep-onafhankelijk gebaseerde semantiek elk predikaat hoogstens één maal een betekenis hoeft te geven, dit in tegenstelling tot de natuurlijke semantiek waarin elke predikaat voor elk van de voorkomende oproepen geïnterpreteerd moet worden. Bij zware analyses kan dit een heuse kost-besparing betekenen.

Hierbij zou duidelijk moeten worden dat we de natuurlijke semantiek wensen te gebruiken als formalisatie van de concrete uitvoering van een programma, terwijl we de oproep-onafhankelijke semantiek willen gebruiken als basis voor de formalisatie van de eigenlijke programma analyses. Om echter de resultaten van de programma analyses te kunnen relateren tot de concrete uitvoering van het programma, moeten we aantonen dat de semantieken *equivalent* zijn. Uit nader onderzoek blijkt dat deze semantieken over het algemeen niet zomaar equivalent zijn, maar dat dit afhangt van het domein waarmee men deze semantieke instantieert. In Stelling 5.4 in de thesis tonen we aan dat indien het specifieke

domein waarmee deze semantieken geïnstantieerd worden aan bepaalde voorwaarden voldoet, dat deze geïnstantieerde semantieken equivalent zijn. Terwijl deze voorwaarden niet meteen zullen opgaan voor de abstracte domeinen waarmee we onze concrete eigenschappen willen benaderen, zo blijkt echter dat ze wel opgaan voor het domein van de concrete eigenschappen zelf. We verkrijgen dan het volgende schema van geïnstantieerde semantieken, waarbij we het concrete domein $\mathcal{C}$ noemen, en het overeenkomstige abstracte domein $\mathcal{A}$. We gebruiken het symbool $\propto$ om aan te duiden dat een semantiek een correcte benadering is van een andere semantiek.

$$\begin{array}{ccc} \mathcal{S}em_M(\mathcal{C}) & \overset{c}{\Longleftrightarrow} \\ & \text{(Th. 5.4)} \\ \propto\downarrow & & \propto\downarrow \\ \mathcal{S}em_M(\mathcal{A}) & & \mathcal{S}em_{M\bullet}(\mathcal{A}) \end{array} \quad \mathcal{S}em_{M\bullet}(\mathcal{C})$$

Dit schema kan men als volgt opvatten. Indien men de concrete eigenschappen van een programma beschrijft aan de hand van een domein $\mathcal{C}$, en indien dit domein aan de equivalentievoorwaarden voldoet, dan is de natuurlijke semantiek van een Mercury programma m.b.t. deze eigenschappen equivalent met de oproep-onafhankelijke gebaseerde semantiek van dit programma. Dit heeft als gevolg dat zowel $\mathcal{S}em_M(\mathcal{A})$ alsook $\mathcal{S}em_{M\bullet}(\mathcal{A})$ correcte benaderingen zijn van de concrete uitvoering van het programma, voorgesteld door $\mathcal{S}em_M(\mathcal{C})$. Natuurlijk is voor ons vooral $\mathcal{S}em_{M\bullet}(\mathcal{A})$ van belang.

Merk op dat dit schema niets zegt over de equivalentie tussen $\mathcal{S}em_M(\mathcal{A})$ en $\mathcal{S}em_{M\bullet}(\mathcal{A})$. Voor bepaalde domeinen die wij in deze thesis ontwikkelen zal het zelfs zo zijn dat $\mathcal{S}em_M(\mathcal{A})$ een benadering is voor $\mathcal{S}em_{M\bullet}(\mathcal{A})$, wat dus betekent dat men met $\mathcal{S}em_{M\bullet}(\mathcal{A})$ nauwkeurigere resultaten kan verkrijgen, wat dan ook weer het voordeel van het gebruik van $\mathcal{S}em_{M\bullet}$ benadrukt voor deze abstracte domeinen.

5 Geheugenstructuren

Om aan geheugen herbruik te kunnen doen hebben we een formele taal nodig om de basiseenheid aan te kunnen spreken: namelijk de geheugencellen die men inderdaad wenst te herbruiken. Hiervoor voeren we de notie van *geheugenstructuur* in (Eng. *data structure*). Een geheugenstructuur stelt niet één enkele geheugencel voor maar wel een verzameling van geheugencellen die gebruikt worden om een volledige term in het computer geheugen voor te stellen. Zo is de geheugenstructuur van de term waaraan de variabele JackRecord gebonden is in Figuur 1 (Deel a) de verzameling van 4 geheugencellen die nodig zijn om deze term volledig voor te stellen: de 3 cellen om de argumenten van het functiesymbool

employee/3 voor te stellen, en de geheugencel waarin de naam "Jack␣Newman" bewaard wordt.

Omdat in Mercury elke term beschreven wordt door een type, zo kunnen we de beschrijving van dit type gebruiken om elk van de delen van de geheugenstructuur van een term aan te wijzen.

In Mercury[1] zou men de volgende types kunnen definiëren:

```
t ---> f(int); g(int,int).
list ---> [] ; [t|list]
```

Een term van type t is een term met functiesymbool f/1 of een term met functiesymbool g/2. Beide vormen nemen getallen als argumenten (type int). Een lijst wordt gedefinieerd als zijnde ofwel de ledige lijst ([]), ofwel een structuur met als eerste argument een term van type t en als tweede argument opnieuw een lijst. Men kan deze type-definities vertalen naar grafes, waarop men vervolgens de functiesymbolen kan gebruiken om paden op te bouwen waarmee men dan individuele stukken geheugenstructuren kan aanwijzen.

Indien we aannemen dat een variabele A tijdens de uitvoering van een programma gebonden wordt aan een term [f (3) | [f (4) | [g (5,5) | []]]] van het bovengedefinieerd type list, dan kunnen we bijvoorbeeld volgende geheugenstructuren van A aanspreken:

- A^{ϵ} stelt de volledige geheugenstructuur van A voor;
- $A^{([|],1)}$ selecteert langs het eerste argument van functor $[|]$ de geheugenstructuur van de term f(3) voor;
- $A^{([|],2)}$ selecteert langsheen het tweede argument van functor $[|]$ de geheugenstructuur van sublijst [f (4) | [g (5,5) | []]] voor:
- met $A^{([|],2)\cdot([|],2)}$ maken we twee stappen in de geheugenstructuur van A, en verwijzen we naar de sublijst [5 | []] van A.

Op gelijkaardige manier kan men elke geheugenstructuur van elke term waaraan een variabele tijdens de uitvoering van een programma gebonden wordt aanspreken. We noemen de paden, in het voorbeeld ϵ, $([|],2)$, $([|],2)\cdot([|],2)$ etc., die we hierbij opbouwen *selectoren*. Variabelen gecombineerd met zulke paden noemen we *concrete geheugenstructuren* omdat we hiermee het geheugen kunnen beschrijven zoals het tijdens de uitvoering van een programma gebruikt wordt.

Indien men een programma echter bekijkt zonder het uit te voeren, zo kan men niet op voorhand weten aan welke termen de variabelen gebonden zullen worden. Indien men de voorgaande voorstelling van geheugenstructuur blijft hanteren, bestaat er het gevaar dat men oneindige paden opbouwt. Om dit te verhelpen gaat men een equivalentie-relatie opstellen voor selectoren op basis van

[1] Hier hebben we het dan niet over de formele syntax zoals hier boven voorgesteld waar deze informatie impliciet aanwezig gezien wordt

het type van het stuk geheugen dat een selector bij een variabele aanwijst, alsook op basis van de hiërarchie van selectoren. Zonder in detail hierop in te gaan, zo kan men bij de variabele A van type list tijdens de analyse van het programma spreken over de volgende geheugenstructuren:

- $A^{\overline{\epsilon}}$: de volledige geheugenstructuur waar A naar zal kunnen wijzen tijdens de uitvoering van het programma *of* elke sublijst van A;
- $A^{\overline{([],1)}}$: het eerste argument van de lijst waar A tijdens de uitvoering naar zou kunnen wijzen.

Bij deze notatie zal $\overline{([],2)}$ als equivalent beschouwd worden met $\overline{\epsilon}$ voor termen van het type list omdat beide selectoren verwijzen naar *al* de sublijsten van list-termen, waaronder de hoofdlijst zelf ook.

Het is duidelijk dat men met deze notatie niet één specifieke geheugenstructuur van een variabele beschrijft, maar meteen een hele verzameling geheugenstructuren. We spreken hier dan ook over *abstracte geheugenstructuren* en gebruiken ze als benadering voor de concrete geheugenstructuren die daadwerkelijk tijdens de uitvoering van een programma opgebouwd worden.

In de thesis werken we het domein van concrete geheugenstructuren en het domein van abstracte geheugenstructuren nauwkeurig uit en geven ze een traliestructuur. In de volgende paragraaf gebruiken we deze domeinen om de actieve concrete geheugenstructuren van een programma te benaderen door de actieve abstracte geheugenstructuren uit het programma af te leiden op basis van programma analyse.

6 Basis Analyses

Beschouw het volgende code-fragment waarin we gebruik maken van het predikaat updateSalary/3 dat we voorheen gedefinieerd hebben.

Voorbeeld 2

```
E = employee("Jack␣Newman", 19490319, 40000),
EL = [E | []],
updateSalary(E, 42000, NE),
NEL = [NE | EL]
```

Indien we deze code concreet uitvoeren, dan maken we hierin een employee/3 term aan en kennen het toe aan de variabele E. Deze variabele E wordt gebruikt in een lijst-term waarnaar EL zal verwijzen. Door de oproep naar updateSalary/3 wordt er een nieuwe term NE aangemaakt, die dan wederom tesamen met EL gebruikt wordt om een lijst van twee werknemers-termen op te bouwen, namelijk

NEL. We hebben gezien dat updateSalary/3 een mogelijkheid heeft tot geheugenherbruik, en de vraag is dus: mogen we bij deze oproep het geheugen waar E naar verwijst gebruiken om NE op te laten bouwen? Puur intuïtief zien we dat het antwoord natuurlijk "*neen*" is, maar hoe kunnen we dit antwoord formeel aantonen?

Bij nader onderzoek stellen we vast dat hier twee factoren een rol spelen: (1) op het moment van de updateSalary-oproep kan de geheugenstructuur waar E naar verwijst ook via EL worden aangesproken (met name als $EL^{([],1)}$) — men zegt dat E en EL geheugenstructuur *delen* (Eng. *data structure sharing*) — en (2) EL wordt na deze oproep duidelijk nog aangesproken (bij de constructie van NEL). Het feit dat EL nog gebruikt wordt betekent dat diens geheugenstructuren intact moeten blijven en omdat EL en E gedeelde geheugenstructuren bevatten geldt dit ook voor E.

Hierdoor wordt duidelijk dat de actieve geheugenstructuren van een programma enerzijds bepaald worden door het al dan niet gebruik van de voorkomende variabelen (EL komt na de oproep van updateSalary dus is diens gehele geheugenstructuur op dat moment actief), en anderzijds door de gedeelde geheugenstructuren die tijdens de uitvoering van het programma kunnen voorkomen (E en EL bevatten gedeelde geheugenstructuren, EL is actief, dus is E ook actief op het moment van de oproep in kwestie). Aan de hand van deze twee elementen kan men voor elke literal de verzameling van actieve geheugenstructuren opbouwen. Bij een concrete uitvoering verkrijgt men de actieve concrete geheugenstructuren, terwijl men bij een analyse actieve abstracte geheugenstructuren zal verkrijgen als benadering van deze actieve concrete geheugenstructuren.

Terwijl men het voorkomen van de variabelen bijna puur op syntactische basis kan bepalen, zo vergt het afleiden van de gedeelde geheugenstructuren een volledige programma analyse. Op basis van het domein van de geheugenstructuren ontwikkelen we het domein van de *gedeelde geheugenstructuren*. Het voornaamste verschil is dat dit domein niet is opgebouwd uit eenvoudige geheugenstructuren, maar uit koppels van deze geheugenstructuren. Zo kunnen we de gedeeldheid van E en EL uitdrukken als $\left(E^{\epsilon} - EL^{([],1)}\right)$ in het concrete geval, en als $\left(E^{\overline{\epsilon}} - EL^{\overline{([],1)}}\right)$ in het abstracte geval. Dit resulteert in het domein van concrete gedeelde structuren enerzijds, en het domein van abstracte gedeelde structuren anderzijds. In de thesis tonen we aan dat voor het concrete domein de equivalentievoorwaarden opgaan van Stelling 5.4, wat betekent dat de natuurlijke semantiek geïnstantieerd met dit domein theoretisch equivalent is met de oproep-onafhankelijke semantiek geïnstantieerd met ditzelfde domein. Als gevolg hiervan definiëren we de programma analyse voor het afleiden van de abstracte gedeelde structuren (in het Engels noemen we dit de *structure sharing analysis*) simpelweg als de oproep-onafhankelijke semantiek geïnstantieerd met het domein van abstracte gedeelde geheugenstructuren. Gezien de equivalentie

voor het concrete domein verkrijgen we automatisch dat deze analyse correcte resultaten oplevert t.o.v. de geheugen situaties zoals ze verkregen worden tijdens het eigenlijke uitvoeren van de geanalyseerde programma's.

Door op gepaste wijze de gegevens over de gedeeldheid van de geheugenstructuren te combineren met de informatie over de al dan niet gebruikte variabelen binnen een bepaald predikaat kan men voor elke beschrijving van een oproep naar dat predikaat de actieve abstracte geheugenstructuren bepalen. Deze stap noemen we de analyse naar de actieve geheugenstructuren van een programma (Eng. *liveness analysis*). Uit deze formulering blijkt wel dat men de actieve geheugenstructuren van een predikaat enkel kan bepalen indien men weet hoe het opgeroepen wordt, dus indien men een beschrijving heeft van die oproep. Hier komen we in de context van modules nog op terug.

Op basis van de kennis over de actieve geheugenstructuren binnen een programma kan men tenslotte de mogelijkheden tot geheugenherbruik nagaan. De plaats bij uitstek om dit te doen is bij deconstructie unificaties. Bij deze unificaties wordt een term ontleed, en nieuwe verwijzingen aangemaakt naar elk van de argumenten van de ontlede term. Indien blijkt dat bij deze deconstructie de variabele die naar de te ontleden term verwijst de laatste verwijzing is naar deze term, dan kan men hieruit besluiten dat de geheugencellen overeenstemmend met deze term herbruikt mogen worden. Dit kan men uit de informatie over de actieve geheugenstructuren na gaan door te controleren of de geheugenstructuur overeenstemmend met deze term al dan niet actief is: als hij niet actief is, is hij afgedankt, en zodus kan hij herbruikt worden na de deconstructie.

We illustreren dit a.d.h. van het gebruik van updateSalary zoals gedefinieerd in Voorbeeld 1. De oproep naar updateSalary zoals in Voorbeeld 2 kan men als volgt beschrijven: vermits $E^{\overline{\epsilon}}$ gedeeld is met $EL^{\overline{([],1)}}$, en $EL^{\overline{\epsilon}}$ is een actieve gegevenstructurur op dat moment, zo is $E^{\overline{\epsilon}}$ dus actief, wat we naar de context van de formele variabelen van updateSalary kunnen vertalen naar het feit dat de variabele EmployeeRecord verwijst naar een actieve geheugenstructuur. De eerste unificatie is een deconstructie waarin de term van EmployeeRecord ontleed wordt. Gezien echter de overeenstemmende gegevenstructuur actief is in deze oproep, kan men niets herbruiken. Wat ons intuïtief antwoord op de vraag "*kan men herbruiken?*" formeel bevestigt.

Laat ons nog een tweede voorbeeld beschouwen.

Voorbeeld 3

```
E = employee("Jack␣Newman", 19490319, 40000),
updateSalary(E, 42000, NE),
EL = [NE]
```

Hier wordt een employee-term aangemaakt en toegekend aan E, waarop meteen updateSalary wordt opgeroepen. Na deze oproep wordt E zelf nergens meer

in het programma gebruikt. Op het moment van de oproep zijn er geen gedeelde structuren, en ook geen actieve geheugenstructuren, wat betekent dat updateSalary als laatste toegang heeft tot de term waar *E* naar verwijst. Vertaald naar de context van de formele argumenten van updateSalary betekent dit dat er geen gedeelde geheugenstructuren zijn bij deze oproep, alsook geen actieve geheugenstructuren. Op het moment van de deconstructie blijkt de geheugenstructuur van EmployeeRecord dus niet tot de actieve geheugenstructuren te behoren, waaruit de analyse mag besluiten dat het geheugen dat overeenstemt met de ontlede term herbruikt mag worden, met name de drie cellen zoals aangegeven in Figuur 2 (Deel b). Gezien de deconstructie meteen opgevolgd wordt door een constructie is het vanzelfsprekend om de afgedankte geheugencellen meteen te herbruiken voor deze constructie en dus de toekenning van nieuw geheugen uit te sparen.

Het is op deze manier dat men tijdens de analyse naar het *herbruik van geheugenstructuren* systematisch zal nagaan waar geheugenstructuren afgedankt worden, en waar ze kunnen herbruikt worden. Hierbij zou het de lezer wel moeten opvallen dat men altijd een beschrijving van de oproep van een predikaat nodig heeft. Hoe we dit kunnen vermijden wordt in de volgende paragraaf geschetst.

7 Module-gebaseerd Geheugen Herbruik

Een welgekende manier om programma's te structureren, zeker in de context van een team van programma-ontwikkelaars, is het gebruik van modules. Zo ook bij Mercury programma's.

Een Mercury programma wordt dus onderverdeeld in verschillende modules. In elk van deze modules maakt men een onderscheid tussen de predikaten die door andere modules gebruikt mogen worden, de zogeheten *geëxporteerde predikaten*, en de predikaten die enkel voor intern gebruik dienen. Bij de vertaling van een Mercury programma zal de vertaler één voor één elke module afzonderlijk vertalen. Om echter de nodige controles te kunnen doen wat betreft het juist gebruik van types, mode en determinisme, heeft de vertaler bij het vertalen van een module toch een minimum aan informatie nodig over de modules waarop deze module steunt. Men maakt hiervoor gebruik van *interface bestanden*. Deze bestanden herhalen de belangrijkste informatie van hun overeenkomstige modules opdat deze informatie gebruikt zou kunnen worden bij de vertaling van andere modules. Typisch beperkt zich deze informatie tot de geëxporteerde predikaten.

Vermits we de programma analyses willen laten deel uitmaken van het vertalingsproces moeten we er voor zorgen dat ook onze analyses dit vertalingsmodel volgen: elke module moet afzonderlijk kunnen geanalyseerd worden mist een mogelijk minimaal gebruik aan informatie over de modules waar het eventueel op steunt. Het moeilijke hieraan is dat men predikaten zal moeten analyseren zonder dat men weet hoe ze opgeroepen zullen worden.

Vermits we de gedeelde geheugenstructuren analyse hebben kunnen definiëren als een instantiatie van de oproep-onafhankelijk gebaseerde semantiek, vergt het inpassen van deze analyse in een modulair schema geen verdere aanpassingen. Het wordt echter wel problematisch indien we de actieve geheugenstructuren alsook het herbruik van geheugenstructuren bestuderen. Beide hebben namelijk een inherent oproep-afhankelijk karakter: pas als je weet *hoe* een predikaat opgeroepen wordt, pas dan kan je je actieve geheugenstructuren benaderen, en pas dan kan je op veilige wijze mogelijk geheugenherbruik toelaten. Maar in de aanwezigheid van modules heb je natuurlijk geen beschrijving van de oproep.

Om dit probleem aan te pakken hebben we beslist om als volgt te werk te gaan. Voor elk predikaat voeren we een oproep-onafhankelijke analyse uit naar diens actieve geheugenstructuren. Deze oproep-onafhankelijke analyse is equivalent met een analyse waarbij men aanneemt dat er bij de oproep van een predikaat geen gedeelde geheugenstructuren voorkomen, alsook geen actieve geheugenstructuren. Voor elke literal binnen een predikaat verkrijgen we aldus een beschrijving van welke geheugenstructuren *zeker* actief zullen zijn, losstaand van een specifieke oproep. Deze informatie kunnen we gebruiken in de geheugenherbruik analyse om na te gaan welke structuren in deze ideale omstandigheden herbruikt kunnen worden. Omdat we echter niet weten hoe een predikaat nu juist opgeroepen zal worden, en welke vormen van geheugengebruik nu feitelijk gaan toegepast kunnen worden, hebben we beslist om voor elk predikaat hoogstens twee versies of varianten aan te maken: één versie waarin geen geheugengebruik voorkomt en één versie waarin elk van de gedetecteerde geheugenherbruik mogelijkheden verwezenlijkt wordt. Het spreekt voor zich dat de eerste versie overeenstemt met de versie die men bij normale vertaling zou verkrijgen, terwijl de tweede versie een volledig geoptimaliseerde versie is. De eerste versie heeft als eigenschap dat elke oproep er naar altijd veilig is, terwijl dit bij de tweede versie niet het geval is. Men dient zich dan de vraag te stellen: wanneer mag die geoptimaliseerde versie dan wel opgeroepen worden?

Hiertoe voeren we in Hoofdstuk 10 van de thesis de notie van *herbruik informatie* (Eng. *reuse information*) in. Deze informatie is gekoppeld aan een literal waarbij geheugencellen afgedankt kunnen voorkomen. Dit kan een deconstructie zijn — indien deze deconstructie gevolgd wordt door een adequate constructie spreken we van een mogelijkheid tot *rechtstreeks herbruik* — maar het kan ook een predikaat-oproep zijn waarin geheugen herbruik mogelijk kan zijn — in dat geval spreken we van een mogelijkheid tot *onrechtstreeks herbruik*. De herbruik informatie, bij een rechtstreekse of onrechtstreekse vorm van herbruik, zal al de nodige informatie behelzen die nodig is om de veiligheid van dat herbruik na te kunnen gaan voor een gegeven oproep-beschrijving. Het afleiden van de herbruik gegevens kan op oproep-onafhankelijke wijze gebeuren. Gezien het belang van de veiligheid voor geheugen herbruik gaan we in deze thesis nauwkeurig in op de manier waarop deze gegevens afgeleid kunnen worden, en tonen we zorgvuldig

de veiligheid van de bekomen resultaten aan.

Het resultaat van het analyseren van een module naar het geheugenherbruik binnen diens predikaten bestaat zodoende uit twee delen: (1) een interface bestand dat zowel de oproep-onafhankelijke gedeelde structuren informatie beschrijft voor elk van de geëxporteerde predikaten alsook alle herbruik informatie voor elk van deze predikaten; en (2) een vertaling van de module waarbij voor elk geëxporteerd predikaat mogelijk twee varianten aangemaakt zijn: een altijd veilig op te roepen variant waarin geen herbruik plaats vindt, en een volledig geoptimaliseerde variant die enkel mag opgeroepen worden indien men aan de hand van de bewaarde herbruik gegevens de veiligheid van de oproep kan aantonen.

Bij het analyseren van een andere module volstaat het dan om gebruik te maken van de interface gegevens om ook deze module op de juiste manier te vertalen. We illustreren dit aan de hand van een voorbeeld.

Laat ons een programma beschouwen dat o.a. bestaat uit een module waarin het predikaat uit Voorbeeld 1 gedefinieerd en geëxporteerd wordt, en een module waarin het predikaat create/1 gedefinieerd alsook geëxporteerd wordt:

Voorbeeld 4

```
create(EL) :-
        E = employee("Jack_Newman", 19490319, 40000),
        updateSalary(E, 42000, NE),
        EL = [NE]
```

Zoals u het wellicht al herkend heeft, is het lichaam van create niets minder dan de code van Voorbeeld 3, waarvan we weten dat herbruik mogelijk kan zijn. We zouden dit herbruik ook bij een modulaire analyse willen verkrijgen.

We weten reeds dat updateSalary een mogelijkheid tot (rechtstreeks) geheugenherbruik heeft. Zodus zal de analyse van de module waarin dit predikaat gedefinieerd is twee varianten voor dit predikaat aanmaken: één variant waarin geen herbruik verwezenlijkt wordt, en één variant waarin dit wel gebeurt. De voorwaarde opdat de herbruik-variant gebruikt mag worden is dat het eerste argument van een oproep naar dit predikaat niet naar een actieve geheugenstructuur mag wijzen. Dit wordt door de bijhorende herbruik informatie beschreven[2]. Bij de oproep-onafhankelijke analyse van create zien we nu dat in dat geval updateSalary inderdaad opgeroepen wordt met een eerste argument wijzend naar een afgedankte geheugenstructuur, zodus kunnen we besluiten dat create een onrechtstreekse vorm van herbruik kan toelaten. Als resultaat van de analyse van create zal er een geoptimaliseerde versie van create aangemaakt worden waarin

[2]In deze samenvatting kunnen we echter niet ingaan in de formele details van het domein van herbruik informatie.

de oproep van updateSalary vervangen wordt door een oproep naar diens geoptimaliseerde versie. De analyse zal de bijhorende herbruik informatie hiervoor afleiden.[3]

8 Implementatie en Experimenten

We hebben elk van de hierboven beschreven analyses binnen een bestaande Mercury vertaler geïmplementeerd om zodoende een volledig geheugenherbruik systeem tijdens de vertaling van Mercury programma's te verkrijgen. Opdat dit systeem echter praktisch bruikbaar zou zijn hebben we nog enkele verdere praktische aspecten moeten aanpakken:

- Tot nu toe zijn we er van uit gegaan dat het vinden van een adequate constructie unificatie bij een deconstructie waarin een afgedankte geheugenstructuur ontleed wordt vrij triviaal is. Dit is echter niet zo. De keuze hiervan beïnvloedt namelijk de hoeveelheid geheugenherbruik die men bij een programma kan verkrijgen. In hoofdstuk 11 beschrijven we enkele mogelijkheden om interessante keuzes hierbij te maken.
- Echte Mercury programma's beperken zich niet tot de formele syntax die wij er voor definieerden. Onze analyse kan echter met de weggelaten taalconstructies niets aanvangen. Om de correctheid van de resultaten te garanderen moet de analyse een overschatting gebruiken van wat deze constructies daadwerkelijk als effect kunnen hebben. Deze overschatting kan een desastreuze invloed hebben op de uiteindelijke nauwkeurigheid van de analyse-resultaten. Hieraan moest duidelijk iets gedaan worden. De meest voor de hand liggende aanpak is het voorzien van een manier om deze taalconstructies manueel met de nodige analyse-informatie te voorzien. Ook dat wordt in Hoofdstuk 11 uitvoerig besproken.
- De analyse van programma's moet redelijk efficiënt gebeuren: de analysetijd moet in proportie staan tot de normale vertalingstijd, en toch moeten de analyseresultaten voldoende nauwkeurig zijn. Bij sommige programma's blijkt echter dat de analysetijd serieus uit de hand kan lopen. Om dit te verhelpen definiëren we in deze thesis een manier om analyse resultaten compacter voor te stellen, mits natuurlijk een deel aan nauwkeurigheid in te boeten. Dit versnelt de analyse aanzienlijk, maar het verlies aan nauwkeurigheid blijft beperkt waardoor de meeste interesante vormen van geheugenherbruik vooralsnog afgeleid kunnen worden.

[3] In dit specifieke geval zal herbruik bij elke oproep toegelaten worden omdat de structuur die herbruikt wordt enkel lokaal wordt aangemaakt, en dus niets te maken heeft met de manier waarop het predikaat opgeroepen wordt. Dit betekent dat de analyse elke oproep naar create mag vervangen door een oproep naar de geoptimaliseerde versie, en zodus kan men de niet geoptimaliseerde versie gewoon verwijderen uit het programma.

- Tenslotte voorzien we een manier om afgedankte geheugencellen die niet binnen hun eigen predikaat-definitie herbruikt kunnen worden toch nog te herbruiken. We doen dit door gebruik te maken van een *cache*.

We hebben elk van deze praktische elementen mee geïmplementeerd en konden zodoende het uiteindelijk geheugenherbruik systeem op een aantal kleine maar ook middelgrote voorbeelden uittesten.

Voor de kleine voorbeelden verkrijgen we elke vorm van vooropgesteld geheugenherbruik. Daar hadden we dan ook niet anders van verwacht.

Als middelgrote programma's hebben we gekozen voor een grafische toepassing dat scene beschrijvingen omzet naar gerenderde beelden en een solver waarin een probleem beschreven aan de hand van een aantal beperkingen dient opgelost te worden. Deze toepassingen werden niet speciaal ontwikkeld of veranderd met het geheugenherbruik systeem voor ogen, wat de resultaten natuurlijk des te interessanter maakt.

Over het algemeen kunnen we stellen dat het geheugenherbruik systeem ook bij deze middelgrote voorbeelden heel goede resultaten oplevert. Bij de grafische toepassing kan men het algemeen geheugenherbruik met 50% verminderen, en afhankelijk van het op te lossen probleem kan ook de solver een indrukwekkende hoeveelheid geheugen besparen. Het opmerkelijke echter bij de solver is dat deze gebruik maakt van *niet-deterministische* predikaten, t.t.z. predikaten die meerdere oplossingen kunnen geven. Ook in aanwezigheid van zulke predikaten blijkt het geheugenherbruik systeem mogelijkheden tot geheugenherbruik af te leiden.

9 Optimalisatie Raamwerk

Bij de studie van bovenstaande experimenten komt tot uiting dat onze beperking van hoogstens twee varianten per predikaat aan te maken er helaas voor zorgt dat vele vormen van herbruik uiteindelijk niet toegestaan worden. Dit betekent dat men een lager geheugenherbruik verkrijgt dan wat er potentieel mogelijk is. Dit probleem hebben we trachten aan te pakken in Hoofdstuk 13 waarin we een raamwerk ontwikkelen voor een nieuwe manier om optimalisatie-criteria voor predikaten af te leiden.

Dit raamwerk gelijkt in aanpak aan de achterwaartse analyse voorgesteld door (King and Lu 2002). Zowel in ons raamwerk alsook in de achterwaartse analyse gaat men voorwaarden afleiden op de oproeper opdat een bepaalde literal geoptimaliseerd kan worden. Deze voorwaarden gaat men op gepaste wijze samenvoegen om zo een volledige beschrijving van de mogelijke optimalisaties van een predikaat te verkrijgen. Het verschil tussen beide aanpakken komt tot uiting in de manier waarop dat samenvoegen gebeurt. Bij een backwards analyse gaat men de conjunctie van de onderlinge voorwaarden gebruiken — dit komt overeen met wat we ook in onze implementatie gebruiken: een predikaat mag

enkel een geoptimaliseerd predikaat oproepen indien aan *alle* voorwaarden voldaan is — maar bij ons raamwerk willen we dit versoepelen en gebruiken we de disjunctie. Dit heeft als resultaat dat we bij predikaat-oproepen optimalisatiecriteria afleiden van zodra er een vorm van optimalisatie in het opgeroepen predikaat mogelijk *zou kunnen zijn*. In ons raamwerk verkrijgt men dus een beschrijving van *eventueel mogelijke* optimalisaties, in tegenstelling tot *zeker* mogelijke optimalisaties. Dit verschil heeft als effect dat men de resultaten anders zal moeten gebruiken: men mag ze enkel gebruiken als indicatie, en bij de eigenlijke codegeneratie (waarbij elke vorm van veiligheid gewaarborgd moet blijven) moet men de optimalisatie-voorwaarden nogmaals nagaan.

10 Aanverwant Onderzoek

Alvorens over te gaan tot het eigenlijke besluit geven we hier enkele verwijzingen naar aanverwant onderzoek.

Deze thesis situeert zich binnen het domein van *programma analyse* (Cousot and Cousot 1977) voor *logische programmeertalen* (Cousot and Cousot 1992a; Bruynooghe 1991). We kiezen hier bewust voor een *denotationele aanpak* (Gordon 1979; Allison 1986; Nielson and Nielson 1992) en komen zo terecht bij *denotationele abstracte interpretatie* (Marriott, Søndergaard, and Jones 1994).

We verwezenlijken een geheugenherbruik systeem bij vertaling op basis van twee essentiële analysen: *gedeelde geheugenstructuren analyse*, en *actieve geheugenstructuren analyse*. Deze onderliggende analyses werden ontwikkeld op basis van het werk van Mulkers (1991). Zij beschrijft gelijkaardige analyses voor de logische programmeertaal Prolog. Gezien de grote verschillen tussen Prolog en Mercury was het overnemen van deze analyses naar onze context geen triviale overgang. Mercury is een sterk getypeerde taal, heeft een sterk mode en determinisme systeem, waardoor het zelf afleiden van types, modes en determinisme niet meer nodig is — dit laat toe om de definitie van de programma analyses enigszins te vereenvoudigen. In (Mulkers 1991) gaat men er van uit dat herbruik in aanwezigheid van niet-deterministische code door het uitvoeringssysteem opgevangen wordt. Dit kunnen we in Mercury vermijden door rechtstreeks tijdens de analyse met niet-deterministische code rekening te houden. De basis hiervan werd gelegd in (Bruynooghe, Janssens, and Kågedal 1997).

Op basis van de gedeelde geheugenstructuren analyse en actieve geheugenstructuren analyse ontwikkelen we een geheugenherbruik analyse. Voor zover we weten werd dit in de context van logische programmeertalen enkel in (Debray 1993; Gudjonsson and Winsborough 1993) bestudeerd.

Tenslotte situeert dit werk zich binnen het brede domein van *automatisch geheugenbeheer*. Het blijft echter een feit dat het meeste onderzoek binnen dit domein zich richt naar geheugenbeheer tijdens de *uitvoering* van een programma.

Een goede referentiepunt voor verdere lectuur rond geheugenbeheer tijdens de uitvoering is (Boehm and Weiser 1988; Wilson 1992)

11 Besluit

In deze thesis hebben we een volwaardig automatisch geheugenbeheer systeem ontwikkeld dat tijdens de vertaling van programma's het geheugengebruik nagaat en hiervoor geoptimaliseerde code genereert. Dit systeem werd in de specifieke context van Mercury ontwikkeld, en we hebben er voor gezorgd dat elk van de analyses waarop het steunt wiskundig onderbouwd is waarmee we de veiligheid van de bekomen resultaten kunnen garanderen.

De belangrijkste bijdragen van dit werk zijn het ontwikkelenen van enkele formele semantieken voor Mercury programma's, het formaliseren van de gebruikte analyses op basis van deze semantieken, het aanpassen van de analyses aan een modulaire context, en tenslotte, het implementeren en evalueren van een volledig geheugenbeheer systeem op basis van deze theorie.

Voor zover we weten is dit de meest volledige implementatie van dit type van geheugenbeheer voor een in de praktijk gebruikte programmeertaal. De experimenten met deze implementatie geven voor sommige programma's een geheugenbesparing van rond de 50%.