What is Generic Programming?

Gabriel Dos Reis
Department of Computer Science
Texas A&M University
College Station, TX–77843
gdr@cs.tamu.edu

Jaakko Järvi
Department of Computer Science
Texas A&M University
College Station, TX–77843
jarvi@cs.tamu.edu

Abstract

The last two decades have seen an ever-growing interest in *generic programming*. As for most programming paradigms, there are several definitions of generic programming in use. In the simplest view generic programming is equated to a set of language mechanisms for implementing type-safe polymorphic containers, such as `List<T>` in Java. The notion of generic programming that motivated the design of the Standard Template Library (STL) advocates a broader definition: a programming paradigm for designing and developing reusable and efficient collections of algorithms. The functional programming community uses the term as a synonym for polytypic and type-indexed programming, which involves designing functions that operate on data-types having certain algebraic structures. This paper aims at analyzing core mathematical notions at the foundations of rational approaches to generic programming and library design as reasoned and principled activity. We relate several methodologies used and studied in the imperative and functional programming communities. As a necessary step, we provide a base for common understanding of techniques underpinning generic software components and libraries, and their construction, not limited to a particular linguistic support.

1 Introduction

The notion of "generic programming" has been in use for about four decades, popularized in the '60s with the LISP programming language and its descendents [McC60, ASS84] providing direct support for higher-order functions. Since then, programming techniques and linguistic support for defining algorithms that are capable of operating over a wide range of data structures have been subjects of a large body of work. The notion of *polymorphism* appears to be an essential ingredient of generic programming. In 1967, Christopher Strachey proposed a classification of polymorphism [Str67], based on the linguistic supports present in programming languages. Luca Cardelli and Peter Wegner later refined that classification [CW85], accounting for new language constructs.

Curiously, language features for writing some classes of polymorphic functions and data structures have received more attention than sound programming techniques at the foundation of generic libraries. In fact, generic programming (as usual with successful programming paradigms) is often equated with language features. It is not uncommon to see definitions of "generic programming" that are more or less crafted to mean what the specific programming languages under consideration support [BJJM99]. Similarly, much of the conventions and practice of generic programming in the context of C++ [ISO03, Str00] is shaped by the template system of C++. It is thus difficult to objectively define generic programming without a bias to a particular programming language over others. But if we want to think of generic programming as a principled, reasoned activity, such a language independent understanding is necessary. Consequently, this paper will not focus on language features as the subject of study. The reader interested in a comparison of mainstream programming language features for generic programming is referred to the report of Ronald Garcia *et al.* [GJL+03]. To avoid being lost in the twists and turns of the "empty set theory" we illustrate our ideas and claims with extensive examples written in concrete programming languages, in particular, C++ [ISO03, Str00], Haskell [PJ03], and Scheme [R5R98]. The list of programming languages used in this paper is kept short to avoid distraction. Of course, we hope that the reader would translate or re-express our examples in his or her own favorite programming languages.

Our long term goal is to develop useful theories of generic programming, to better understand and advance the practice of generic programming as a principled activity. This paper reports work in progress along this path, starting from analyzing and relating several notions of generic programming.

It is good to have theories that clarify practice. Good theories, however, are not those that simply rehash common knowledge. Good theories help predict and conquer unexplained and/or unexplored territories. For example, Newton's theory of gravitation was good because it clarified practices and beliefs of the time *but also* helped predict eclipses within reasonable precision. The theories of relativity developed by Einstein were good because they explained facts that left physicists perplexed, and took up where Newton's theory was defeated in predictions. From empirical sciences,

Submitted to LCSD'05

one can observe that useful theories are falsifiable. That is, they can be confronted with hard data from the world. Similarly, we posit that useful theories that help gain better understanding of generic programming should be confronted with practices from the real world. The theories are not the goals in themselves, they are means by which we seek to have better understanding. Also, care must be exercised so as not to confuse theories with realities in interpretations.

As its main contribution, this paper shows how different approaches to generic programming can be explained within the same mathematical framework, leaning on category theory. We note that the connection between category theory and generic programming in functional programming languages has been well established — many generic algorithms draw their motivation from categorial notions. A novelty of this paper is the establishment of similar connections for generic programming approach as pioneered by Alexander Stepanov, David Musser and their collaborators (at the foundation of the STL), which arises largely from a practical perspective of organizing generic software components for increased reusability. The latter approach builds on low level language features — driven by efficiency considerations — much more so than the other approaches to generic programming. As a result, however, proving properties of and reasoning about STL generic algorithms is difficult. We believe a stronger connection to a formal model of generic programming will aid in this respect, guiding the development of generic libraries, and program manipulation tools for them.

2 Background

Generic programming has been approached from various angles in both the functional programming and imperative programming communities. We identify two main schools of thought:

1. the "gradual lifting of concrete algorithms" discipline as first described by David Musser, Alexander Stepanov, Deepak Kapur and collaborators;
2. a calculational approach to programming, the foundations of which were laid by Richard Bird and Lambert Meertens.

The first school defines the discipline of generic programming essentially as follows: start with a practical, useful, algorithm and repeatedly abstract over details; at any stage of the gradual abstraction, the "generic" version of the algorithm shall be such that when instantiated it shall match the original algorithm both in semantics and efficiency. The gradual lifting stops when these conditions cease to hold. Quoting Musser and Stepanov [MS88]:

> By generic programming, we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parameterized procedural schemata that are completely independent of the underlying data representation and are derived from concrete efficient algorithms.

The requirement of abstract specification independent of the actual data representation is fundamental for two reasons: 1) it is at the basis of substitution of one datatype interface for another when they are similar; and 2) it allows for classification of similar interfaces based on their efficiency. For example, the linear search function `find()` of the Standard Template Library [SL94] works on iterators coming from either a linked-list or an input stream because they provide similar interfaces for increment and value-fetching. However, `binary_search()` is defined only for forward iterator interfaces.

The second school of thought in generic programming has its root in the initial algebra approach to datatypes as advocated by Joseph Goguen and collaborators [GTWW77, TWW82] and a calculational approach to program construction [Bir87, Mee86]. Category theory is an essential tool in this setting. In "Generic Programming — an Introduction" [BJJM99], Roland Backhouse *et al.* stated:

> we introduce another dimension to the level of abstraction in programming languages, namely parameterization with respect to classes of algebras of variable signature.

In this approach, also referred to as *datatype generic programming*, structures of datatypes are parameters of generic programs. Datatype generic programming [JJ96, JJ97, BJJM99, Hin00, Hin04] has had a strong focus on regular datatypes essentially described by algebras generated by the functors *sum*, *product* and *unit*. Algorithms written for those functors can then operate on any inductive datatype, and are thus inherently very generic. Indeed, a fairly large class of generic algorithms can be defined in this manner, such as structural equality, serialization/deserialization, zips, folds, and traversals.

The Musser–Stepanov style of generic programming emphasizes *concept analysis*, the process of finding and establishing the important classes of concepts that enable many useful algorithms to work. Programmers then explicitly define correspondence from their datatypes to those classes of concepts. A thesis of this paper is that concept analysis is a way of looking for functors that capture common structures. We can see that the two definitions of generic programming are fundamentally very close to each other, but the emphasis in each view is on different aspects: one focusing on a particular structural algebra for datatypes and the algorithms defined in terms of that algebra, whereas the other on finding and classifying classes of algebras based on some notions of efficiency.

Finally, we can observe that while both methodologies have an underlying theoretical language-independent model, C++ has become the dominating platform for the Musser–Stepanov style[1], whereas Haskell and its variants are the almost exclusive tool for data-type generic programming.

3 Using category theory

Category theory is a branch of mathematics originally developed as a language to unify and abstract over many structure and proof patterns in Algebraic Topology. Category theory — also occasionally referred to as "abstract nonsense" or "the

[1] Though Musser's and Stepanov's early work on generic programming was in the context of Scheme and Ada.

theory of empty set" — has found an unreasonably effective application in Computer Science. The theoretical core ideas of the categorial approach to datatypes and generic functions go back at least to Goguen and collaborators [GTWW77].

3.1 Elementary notions

This section recalls some basic notions of category theory and establishes vocabulary used in the rest of the paper. We have kept the load of jargon to the minimum; the reader interested in further development of category theory might advantageously consult the standard textbook of Saunders Mac Lane [ML01]. Within the discussion, we include examples of how the categorial notions become manifest as idioms and patterns in practical programming.

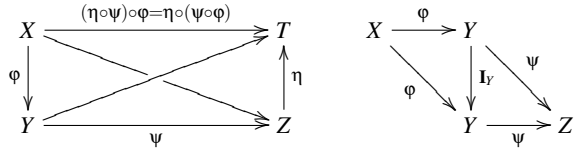
3.1.1 Categories

A *category* $\mathcal{C}$ is a collection of *objects* and *arrows* (also called *morphisms*) between objects with three fundamental operations:

1. Every arrow φ in $\mathcal{C}$ is associated with two objects:
   - its *source* $\mathrm{dom}\,\varphi$, an object of $\mathcal{C}$, and
   - its *target* $\mathrm{cod}\,\varphi$, also an object of $\mathcal{C}$.

   Thus, an arrow is often written as $\varphi : X \to Y$, where X is the source and Y the target.
2. Every object X in $\mathcal{C}$ is associated with a distinguished arrow $\mathbf{I}_X : X \to X$, called the identity arrow of object X.
3. For two composable arrows $\varphi : X \to Y$ and $\psi : Y \to Z$ in $\mathcal{C}$, the composition $\eta = \psi \circ \varphi : X \to Z$ is again an arrow in $\mathcal{C}$.

   Furthermore, the composition operator must be associative and admits the identity arrow as unit, which diagrammatically reads



The collection of arrows from an object X to an object Y is called the *hom-set* from X to Y and written $\hom_{\mathcal{C}}(X,Y)$. The subscript is used to emphasize the category under consideration.

3.1.1.1 Examples

Small sets Our first example of a category is **Set** whose objects are sets and arrows are the usual total functions between sets.

Complete partial orders Recall that a *partial order* $\preccurlyeq$ on a set X is a binary relation on X that is reflexive, transitive and antisymmetric. A set equipped with a partial order is said a *partially ordered set* or *poset* for short. For example, the set $\mathbb{N}$ of natural numbers equipped with the relation "divides" is a poset. A function f from a poset $(X, \preccurlyeq_X)$ to a poset $(Y, \preccurlyeq_Y)$ is said *monotonic* if $f(x_1) \preccurlyeq_Y f(x_2)$ whenever $x_1 \preccurlyeq_X x_2$. An ω-*chain* in a poset X is a sequence $x : \mathbb{N} \to X$ such that $x_i \preccurlyeq x_{i+1}$. A poset in which every ω-chain has a least upper bound is called an ω-*complete poset*. An ω-complete poset with a least element is said to be an ω-*complete pointed poset*. For example, the power set 2^A of a set A is an ω-complete pointed poset when equipped with inclusion as partial order.

A *continuous* function between two posets is a monotonic function that sends the least upper bound of an ω-chain to the least upper bound of the image of the chain. The collection **CPO** of ω-complete pointed posets is a category where the arrows are continuous functions; $\mathbf{CPO}_\perp$ is a **CPO** with a least element.

3.1.2 Initial and terminal objects

An object $\mathfrak{i}$ is called *initial* in a category $\mathcal{C}$ if, for every object X in $\mathcal{C}$, the hom-set $\hom_{\mathcal{C}}(\mathfrak{i}, X)$ is a singleton. Dually, an object $\mathfrak{t}$ is said to be *terminal* if for every object X in $\mathcal{C}$, the hom-set $\hom_{\mathcal{C}}(X, \mathfrak{t})$ is a singleton. A category can admit at most one initial (resp. terminal) object, up to isomorphism.

3.1.2.1 Examples

In **Set**, the empty set $\mathbf{0}$ is initial. On the other hand, every singleton $\mathbf{1}$ is terminal.

3.1.3 Functors

Categories are not very interesting by themselves; what is interesting about them is *what* is happening in or between them, *e.g.* functors, etc. that we will define shortly. When studying structures, the first natural thing one usually does is to look for properties that remain unchanged over similar structures. For categories, that means properties that remain unchanged through the composition operator in a class of structures.

A *functor* F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a morphism of categories; it consists of two parts:

1. An *object function* which assigns an object $F(X)$ in $\mathcal{D}$ to every object X in $\mathcal{C}$;
2. An *arrow function* that assigns an arrow $F(\varphi) : F(X) \to F(Y)$ in $\mathcal{D}$ to every arrow $\varphi : X \to Y$ in $\mathcal{C}$ such that
   - the identity arrow is sent to the identity arrow, *i.e.*,
     $$F(\mathbf{I}_X) = \mathbf{I}_{F(X)}$$
     for every object X in $\mathcal{C}$,
   - two composable arrows $\varphi : X \to Y$ and $\psi : Y \to Z$ are sent to composable arrows and the property
     $$F(\psi \circ \varphi) = F(\psi) \circ F(\varphi)$$
     holds.

We will say that $F(\varphi)$ is the *lift* of the arrow φ by F.

3.1.3.1 Examples

Identity functor A ubiquitous functor is the identity functor **I**. Both its object function and arrow function yield their arguments unchanged.

Constant functor Any object A in a category $\mathcal{C}$ gives rise to a functor $\mathbb{A}$ as follows: the object function sends all objects to A, and the arrow function sends all arrows to the identity arrow of A. In particular, "the" singleton object $\mathbf{1}$ gives rise to the unit functor $\mathbb{1}$.

3.1.4 Multivariate functors

The notion of functor can be generalized to that of *bifunctor*, operating simultaneously on two categories so that the composition law holds component-wise:

$$F(\varphi_2 \circ \varphi_1, \psi_2 \circ \psi_1) = F(\varphi_2, \psi_2) \circ F(\varphi_1, \psi_1).$$

3.1.4.1 Examples

For the purpose of this paper, we will assume that we are mostly working in $\mathbf{CPO}_\perp$. This simplifies the exposition allowing us to talk about least and greatest fixed points, making the connection to algebras and co-algebras less heavyweight. The functor examples given in this section could, however, be defined in a more general setting by universal property, i.e., by singling out specific objects with unique arrows to or from them.

Product functor A commonly used functor is the product functor. Its object function sends two objects X and Y to the object

$$X \times Y = \{(x,y) \mid x \in X, y \in Y\}$$

and its arrow function sends two arrows $\varphi : X \to S$ and $\psi : Y \to T$ to the arrow $\varphi \times \psi : X \times Y \to S \times T$ defined by

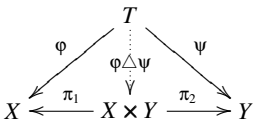
$$(\varphi \times \psi)(x,y) = (\varphi(x), \psi(y)).$$

It can be readily verified that $\times$ indeed is a bifunctor.

The product functor is concretely realized in programming languages in various ways. In C++ for instance, the object function is implemented by the standard library class template `std::pair<X, Y>`. However, there is no predefined arrow function. One can be literally defined as

```
template<class X, class Y, class S, class T>
std::pair<S, T>
lift(const std::pair<X, Y>& p, S f(X), T g(Y))
{
  return std::pair<S, T>(f(p.first), g(p.second));
}
```

Associated with the product functor are the projection combinators π_1 and π_2 leading to the tupling combinator $\triangle$ that makes the following diagram commute



for any pair of arrows $\varphi : T \to X$ and $\psi : T \to Y$.

In code, the tupling combinator would read

```
template<class T, class X, class Y>
std::pair<X, Y> tuple(T t, X f(T), Y g(T))
{
   return std::pair<X, Y>(f(t), g(t));
}
```

Sum functor Yet another commonly used functor is the discriminated union. It takes objects to *tagged pairs*

$$X + Y = \{0\} \times X \cup \{1\} \times Y \cup \{\perp\}$$

and arrows to arrows *defined by case analysis*

$$(\varphi + \psi)(\perp) = \perp$$
$$(\varphi + \psi)((0,x)) = (0, \varphi(x))$$
$$(\varphi + \psi)((1,y)) = (1, \psi(y))$$

where a pattern matching is done as follows: if the argument is junk, then it is returned untouched; if the argument was built from an element of the first component then it is extracted, given to the first arrow and the result is packaged back into the first component; otherwise if the argument was built from an element of the second component then it is extracted, given to the second arrow and the result is packaged back into the second component.

The above behavior takes lots of words to describe but very few symbols to define in Haskell

```
data Either a b = Left a | Right b
eitherLift ::
  (a -> c) -> (b -> d) -> Either a b -> Either c d
eitherLift f g (Left x)  = Left (f x)
eitherLift f g (Right y) = Right (g y)
```

Discriminated unions are idiomatically expressed in languages without built-in pattern matching as instances of the Visitor Design Pattern [GHJV94]. In C++ for example, using this scheme we define a base class `Either` with derived classes `Left` and `Right`. A class `EitherVisitor` that can visit classes derived from `Either` is also needed.

```
template<class X, class Y> class Either;
  template<class X, class Y> class Left;
  template<class X, class Y> class Right;

template<class X, class Y>
struct EitherVisitor {
  virtual void visit(const Left<X, Y>&) = 0;
  virtual void visit(const Right<X, Y>&) = 0;
};

template<class X, class Y>
struct Either {
  virtual ~Either() { }
  virtual void accept(EitherVisitor<X, Y>& v) const = 0;
};

template<class X, class Y>
struct Left : Either<X, Y> {
  const X& x;
  Left(const X& x) : x(x) { }
  void accept(EitherVisitor<X, Y>& v) const
    { v.visit(*this); }
};
```

```
template<class X, class Y>
struct Right : Either<X, Y> {
  const Y& y;
  Right(const Y& y) : y(y) { }
  void accept(EitherVisitor<X, Y>& v) const
    { v.visit(*this); }
};
```

The code has a fair amount of boilerplate to simulate pattern matching. Now, the lift mapping itself can be defined as

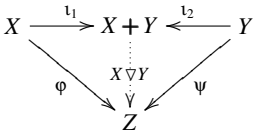
```
template<class X, class Y, class S, class T>
const Either<S, T>
lift(const Either<X, Y>& e, S f(X), T g(Y))
{
   typedef S (*F)(X);
   typedef T (*G)(Y);
   struct Impl : EitherVisitor<X, Y> {
      F f;
      G g;
      const Either<S, T>* value;
      Impl(F f, G g) : f(f) g(g), value() { }

      void visit(const Left<X, Y>& e)
      {
         value = left<S, T>(f(e.x));
      }
      void visit(const Right<X, Y>& e)
      {
         value = right<S, T>(g(e.y));
      }
   };

   Impl vis(f, g);
   e.accept(vis);
   return *vis.value;
}
```

We use helper functions `left<S, T>()` and `right<S, T>()` for allocating objects of the obvious types. The code is undoubtly more involved than the corresponding few lines in Haskell (or ML). It is not intended as a translation of Haskell to C++, but as illustration of both basic categorial constructs and common techniques used in languages lacking direct support for pattern matching.

Dually to the case of product, the sum functor comes with two injection combinators ι_1 and ι_2 and a conflating combinator ∇ making

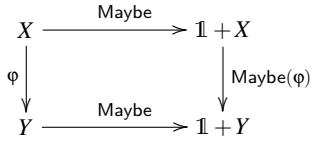


a commutative diagram, for any pair of arrows $\varphi : X \to T$ and $\psi : Y \to T$.

In code, the destruction combinator is typically given by case analysis (because its domain is a discriminated union).

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

The Maybe **functor** It is the functor $\mathbb{1} + \mathbf{I}$ whose action is described diagrammatically as



Conceptually, it describes the type of objects that may hold values of another datatype or nothing.

3.1.5 *Algebras and co-algebras*

In this section we consider only endofunctors, *i.e.*, functors with identical sources and targets.

3.1.5.1 *Algebras*

The notion of algebra generalizes that of Σ-algebra from the theory of Universal Algebra [Coh81] where an algebra can be thought of as interpretation of a collection of function symbols, and the structures of their domains are given by the functor.

Given an endofunctor F of a category $\mathcal{C}$, an arrow of the form

$$\alpha : F(X) \to X$$

is called an *F-algebra* — written $(\alpha, X)_F$ or simply (α, X) when the functor is understood from context — and the object X is its *carrier*.

In $\mathbf{CPO}_\perp$ for example, if one thinks of a polynomial functor as describing a structure X together with operation symbols, then an algebra appears as an interpretation by case analysis.

Example The Haskell datatype

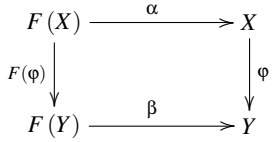
```
data Nat = Zero | Succ Nat
```

is a Maybe-algebra, because the above definition introduces the operation `Zero` ϒ `Succ` where

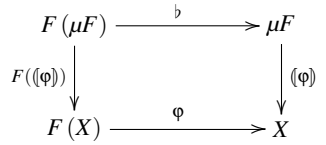
```
Zero :: Nat        -- can be thought as Zero :: 1 -> Nat
Succ :: Nat -> Nat -- successor operation
```

Here we would like to interpret `Zero` as the natural number 0, and `Succ` as the operation that yields the successor of a natural number. Of course, that is not the only possible interpretation; but among all possible interpretations, there is a distinguished one. We make that idea more precise in the following paragraphs.

Given an endofunctor F on a category $\mathcal{C}$ and two F-algebras (X, α) and (Y, β), an arrow $\varphi : X \to Y$ that makes



a commutative diagram, *i.e.*, $\varphi \circ \alpha = \beta \circ F(\varphi)$, is called an *F-algebra homomorphism*. The collection $\mathbf{Alg}(F)$ of F-algebras can be readily seen to form a category where the arrows are the F-algebra morphisms. The initial object $(\mu F, \flat)$ of that category, when it exists, is called the *initial F-algebra*. It has the distinguishing characteristic that given any F-algebra (φ, X) there is unique F-algebra homomorphism — written $(\!|\varphi|\!)$ — from μF to X making the diagram



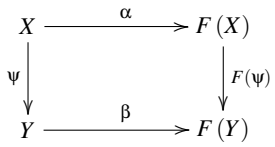
commutative. The arrow $(\!|\varphi|\!)$ is said to be the *catamorphism* of φ. Examples of catamorphisms will be given in §3.2.1

3.1.6 Coalgebras

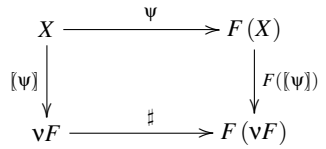
A *coalgebra* is the dual notion of an algebra, *i.e.*, an arrow of the form

$$\alpha : X \to F(X)$$

which we will denote by $[\alpha, X]_F$. One can also define the notion of *F-coalgebra homomorphism* which is an arrow $\psi : X \to Y$ that makes the diagram



commute for any pair of F-coalgebras α and β. The collection $\mathbf{CoAlg}(F)$ of F-coalgebras, with F-coalgebra homomorphisms as arrows, is a category. The terminal object $(\nu F, \sharp)$ of that category, when it exists, is called the *final coalgebra* of the functor F. It is characterized by the fact that given any F-coalgebra $[\psi, X]$, there corresponds a unique F-coalgebra homomorphism from X to νF that makes



a commutative diagram. The F-coalgebra $[\!|\psi|\!]$ is called the *anamorphism* of the arrow ψ.

3.2 Categorial datatypes

3.2.1 Initial datatypes

The initial algebraic approach to datatypes posits that when working in an appropriate category, many abstract data types are nothing but initial algebras of some functor. For example, the usual set of natural numbers as described by the Peano axioms is the initial algebra of the functor Maybe.

The main benefit of viewing datatypes as initial algebras is that an iteration operator over the datatypes, called *fold*, follows for free. That crucial property provides a convenient *implementation* tool and *reasoning* device to capture patterns. In $\mathbf{CPO}_\perp$ for instance, it can be shown that every polynomial functor has an initial algebra, which in fact is its least fixed point.

For example, consider the bifunctor

$$S(T, X) = \mathbb{1} + T \times X = \mathsf{Maybe}(\mathsf{T} \times \mathsf{X}).$$

Its least fixed point with respect to the second argument yields an object parameterized by T

$$Seq(T) = \mathbb{1} + T \times Seq(T)$$

which captures many algebraic aspects of *finite sequences* of values of type T. When viewed as acting on T, it can be thought of as a functor; we will call it the *sequence functor*. A *cons-list* from functional programming practice is an example of such an object. In Haskell, it is defined by

```
data List a = Nil | Cons a (List a)
```

For a fixed T, $Seq(T)$ is the least fixed point of the functor $X \mapsto \mathbb{1} + T \times X$. Computing the length of such list is readily implemented by

```
length :: List a -> Int
length Nil         = 0
length (Cons a as) = 1 + length as
```

where it is apparent that the `length` function is obtained by sending the unit value ($\mathbb{1}$) to 0 and the list constructor `Cons` to the successor operation. That is the essence of catamorphisms, *i.e.*, mapping constructors to functions. Note how that description is an abstract specification of the following C++ algorithm:

```
template<class Forward>
int length(Forward first, Forward last)
{
  int n = 0;
  for (; first != last; ++first)
     ++n;
  return n;
}
```

The fundamental operations of the functor *Seq* are materialized here by

- when to stop or empty sequence $\mathbb{1} \leftrightarrow$ `first == last`;
- next elements of the sequence `++first`.

Then the mapping corresponds to initialization to 0 and incrementation respectively. The act of replacing a signature (here 0 and the successor functions) with a function is the essence of catamorphism, and the basis of polytypic functions. The STL algorithm `accumulate` is the *fold* for sequences, and many other STL algorithms are specializations of it.

3.2.2 Final datatypes

Final datatypes are dual to initial datatypes. They can be modeled as final coalgebras. In the category $\mathbf{CPO}_\perp$, the final coalgebra of a polynomial functor is its *greatest* fixed point. For example, the greatest fixed point of the functor

$$X \mapsto T \times X$$

is the *infinite list* or *stream* of values of types T, characterized by two fundamental operations

$$\texttt{head} : \mathit{Stream}(T) \to T$$
$$\texttt{tail} : \mathit{Stream}(T) \to \mathit{Stream}(T).$$

The C++ standard iterator `ostream_iterator<>` is a genuine example of handles to streams — there is no way to test for "stopping conditions".

The greatest fixed point of the $X \mapsto \mathsf{Maybe}(\mathsf{T} \times \mathsf{X})$ (see §3.2.1) is a *potentially infinite list*. Unlike the case for streams, one can test a potentially infinite list for stopping conditions.

The main difference between initial datatypes and final datatypes is that the former are characterized by constructors whereas the latter are characterized by observers and modifiers.

4 Recursion patterns

The categorial approach to data types makes clear connections between the patterns of "regular" recursive algorithms and those of data types. The most popular being catamorphism, anamorphism and hylomorphism (an anamorphism followed by a catamorphism) [MFP91]. Interestingly, such patterns are essentially present in the Musser–Stepanov approach to structure algorithms, in slightly different forms (iterative mostly) and spelled out differently. Consider the following function template `accumulate` from the STL:

```
template <class Input, class T, class BinOp>
T accumulate(Input first, Input last, T init, BinOp op)
{
  for (; first != last; ++first)
     init = op(init, *first);
  return init;
}
```

This function essentially defines what corresponds to a fold, the general recursion operator for defining catamorphisms, over a `List` functor. Compare this to the typical definition of a fold in, say, Haskell:

```
foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

When `foldr` is called with a mapping for the data constructors list, we get specific catamorphisms. This is directly visible in the Haskell case: the mapping `z` is applied to lists constructed with `[]`, and `f` to lists constructed with the cons operator `:`. We use `foldr` (instead of `foldl`) because it is the natural iteration operation for the list datatype as defined in Haskell. For example, `foldr (+) 0 a:(b:(c:[]))` gives, after mapping `+` and `0` appropriately, `a+(b+(c+0))`.

In the C++ version, `init` corresponds to `z`, the empty list is denoted by the negation of `first != last`, and `op` is the same as `f`. As an example, in Haskell, the catamorphism `length` for computing the length of a list is obtained by mapping `0` to the empty list, and an increment function to the cons constructor:

```
length ls = foldr (1+) 0 ls
```

Analogously, the C++ `length` function can be written in terms of `accumulate` as follows:

```
struct incrementor {
  template<class X, class Y>
  X operator()(X x, const Y& t) const { return x + 1; }
};

template <class In>
int length(In first, In last)
{
  return accumulate(first, last, 0, incrementor());
}
```

With the help of a library that provides convenient notation [JPL03], one can simply write

```
template <class In>
int length(In first, In last)
{
  return accumulate(first, last, 0, _1 + 1);
}
```

Many other STL algorithms — `for_each`, `transform`, and `find` to name a few — can be defined as catamorphisms using `accumulate`. The view of a fold as a combinator that defines a traversal, or recursion pattern, for algebras with a particular signature, applies equally well in the context of STL, as it does in the context of Bird–Meertens formalism. However, whereas generalized folds over all regular data types, such as binary trees, are possible in data-type generic programming, this is not the case for STL. For example, `accumulate` is defined only for sequences, not for algebras describing binary trees. As a remedy, STL defines a homomorphism from binary trees (the `map` data structure implemented as red-black trees) to sequences, but this does not enable generic algorithms that truly operate on the structure of the tree. In particular, the homomorphism fixes in-order as the only traversal for STL maps. There are practical consequences of this. For example, copying a STL map to another map with the `std::copy` algorithm exhibits worst case complexity in terms of necessary rotations in the underlying red-black tree. Similarly, the generic `find` algorithm cannot take advantage of the special structure of the tree.

5 Transforming sequences

In line with our "meta" views developed in the opening of this report, we start with the simple idea of transforming a sequence into another one by applying a given function to each element. For concreteness, here is a Scheme routine for that:

```
(define (map function sequence)
  (cond ((null? sequence) nil)
        (else (cons (function (car sequence))
                    (map function sequence)))))
```

That definition assumes the ubiquitous, built-in, Scheme datatype of list to represent a sequence of items. The program fragment inspects its input with the observers

- `null?` to test for an empty sequence;

- `car` to inspect the value of the head of a sequence;
- `cdr` to get to the remaining items in a sequence;

and constructs its output with:

- `cons` to construct a new sequence out of an existing item and a sequence.

These operations seem to be fundamental primitives needed to write the algorithm as a Scheme program. Data constructors (*e.g.* `cons`) are typical to initial algebra treatment of generic datatypes and functions, whereas observers (*e.g.* `null?`, `car`, `cdr`) are defining characteristics of final coalgebras. Consequently, this expression of the transformation function makes a mixture of initial algebras and final coalgebras. Is that mixture essential to capture the algebraic essence of `map`? We will see a purely initial algebraic formulation in §5.1. If not, is that mixture essential to make `map` operate on a wider class of sequence implementations? A fundamentally final coalgebraic definition is given in §5.2 as a C++ function that operates on a wide variety of sequence instances.

The definition of `map` has a direct imprint of the built-in list type — uses of `null?`, `car`, `cdr` and `cons` that have built-in meaning. As is, it is not usable with another incarnation of sequences, say with `vector`s. However, that limitation can be overcome in several ways. One way is to use symbols — *e.g.* `empty?`, `head`, `tail`, `new-seq` and `null` — that can support the abstract operations on a variety of sequence implementations, based on the "data-directed" programming paradigm [ASS84]. In that perspective, their implementations would abstract away the differences in sequence implementations through runtime type-based dispatch. In C++ such an approach could be expressed through overloading or overriding, whereas in Haskell it would take the form of type classes.

Another way of removing the limitation is via higher-order functions, passing the necessary operators as parameters:

```
(define (map fun seq empty? head tail new-seq null)
  (cond ((empty? seq) null)
        (else (new-seq (fun (head seq))
                       (map fun (tail seq) empty?
                            head tail new-seq null)))))
```

This version is fully general and makes no hard-coded assumptions on how the sequence is represented. However, the function may be awkward to use. In particular, every use site of this function must ensure that the right operations are passed along with the right sequence implementations. For example, calling `map` with a list and `vector-ref` will lead to (runtime) error. We see that what we need here is a way of referring to the iteration operator of the *concrete implementation* of the notion of sequence.

This new version of `map`, as well as the first, features several issues in generic programming — accessors as final coalgebras and constructors as initial algebras.

5.1 A slightly different look at `map`

The `map` function is also part of the Haskell Prelude [PJ03] and defined as

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs)  = f x : map f xs
```

The explicit type annotation makes it unambiguous that `map` is defined to work only on Haskell's built-in datatype list. The only genericity achieved here is the variability of the contained element type. However, as we observed in the previous section, the notion of transformation is not restricted to a specific instance of the notion of sequence.

This expression of `map` uses a slightly different approach. Namely, it accesses the building blocks of the input sequence through pattern matching. Therefore it makes essential use of the list data constructors, and is completely defined in terms of those. It can be completely characterized in terms of the initial algebra for list (which really has a stack implementation in most functional programming languages). Consequently, while the definition works on the list implementation of the notion of sequence, it does not work on the Haskell `Array` implementation or any other sequence implementation that does not use the list constructors.

To overcome the use of built-in constructors that tie `map` to a given data type, the Haskell library uses a type class `Functor` as implementation of the general notion of functor, as discussed in §3.1.3:

```
class Functor f where
    fmap :: (a -> b) f a -> f b
```

The idea is that the symbol `fmap` will be applicable to all type constructors for which there are known instance declarations stating that they act like functors. Given such a declaration, a use of `fmap` on a particular concrete sequence is to be made in conjunction with `Functor` instance declarations for that concrete sequence implementation. This situation reminds us of the drawbacks typical of object oriented programming where operations are closely tied to objects (*e.g.* member functions) so that writing N algorithms for P datatypes requires solving $N \times P$ problems.

Applying Stepanov–Musser's methodology to lift `map` to more generic levels, capable of operating over a wider range of data types, requires giving up specific knowledge of the built-in list type. As a consequence, the expression of the idea of sequence transformation seems to become more involved. To what extent are the added complexities intrinsic to `map` as opposed to language artifacts? Is the increase of complexity a sign of useful generality gain?

5.2 Yet another look at `map`

In this section, we look at the expression of the `map` that in the C++ community is known as the standard algorithm `transform`:

```
template<class In, class Out, class Oper>
Out transform(In first, In last, Out out, Oper op)
{
  for (; first != last; ++first)
     *out++ = op(*first);
  return out;
}
```

It is standard, in programming with C++, to represent sequences as pairs of iterators; thus generic sequence algorithms operate on such representations, as laid out in the STL [SL94]. The operations of reading the head of a sequence and moving to the remaining parts are implemented by `*` and `++` operators. The C++ version of `transform` does not use list (sequence) constructors to build the result. Rather, the formulation uses accessors, as if the view is that of *final datatypes*. Consequently, the algorithm can work on all instances of iterators (therefore sequence instances) that provide similar interfaces. The complexity in terms of the number of concrete sequence implementations and concrete transformation implementations is significantly reduced.

6 Limitations

The semi-open interval model used in the STL to represent sequences leaves some data structures out of the picture, most notably circular lists. Similarly, circular list appears to resist initial data type formulations. In fact, circular lists appear to be more amenable to formalization through final coalgebras [Kam83].

What do we gain from the category theory approach to generic programming? Is it effective? What does it explain and what does it predict?

In our view, the categorial approach seeks to capture common algebraic structures, similarities of interfaces as advocated by Dehnert and Stepanov [DS98] (see Section 7). For example, the *fold()* iteration operators are *implementation* tools and *reasoning* devices for capturing traversal and proof patterns common to a class of generic functions [Hut98]. We find the categorial approach as a promising starting point for a theory that can clarify and explain the practice of Musser–Stepanov style generic programming. Moreover, we believe, in accordance with what we state in the introduction of this paper, that the mathematical framework is sufficient for prediction and conquering new grounds as well such as STL in parallel and distributed programming contexts. Along those lines, we mention that libraries and compiler frameworks [RG03] based on the calculational approach, from functional programming perspective, are subjects of active research.

7 Discussion

In this section, we examine, within the mathematical framework in place, the main two approaches to generic programming. The purpose is to identify commonalities and differences in more definite terms.

Dehnert and Stepanov [DS98] advocate maximizing reuse of software components through alikeness identification:

> [...] Breadth of use, however, must come from the separation of underlying data types, data structures, and algorithms, allowing users to combine components of each sort from either the library or their own code. Accomplishing this requires more than just simple, abstract interfaces — it requires that a wide variety of components share the same interface so that they can be substituted for one another. It is vital that we go beyond the old library model of reusing identical interfaces with pre-determined types, to one which identifies the minimal requirements on interfaces and allows reuse by similar interfaces which meet those requirements but may differ quite widely otherwise. Sharing similar interfaces across a wide variety of components requires careful identification and abstraction of the patterns of use in many programs, as well as development of techniques for effectively mapping one interface to another.

Separating data structures from algorithms is key to reducing the complexity of implementing N algorithms for P data structures, as exemplified by the STL. At first sight, that seems to run contrary to the practice of the calculational approach which puts emphasis on iteration operators (folds) intimately associated with recursive data structures. However, it should be observed that once the class of algorithms of interest is identified (*e.g.* sequence algorithms) the iteration operator is also fixed. Other data structures "just" need to have their iteration operators adapted or mapped to the iteration scheme of reference. For example, in the STL all sequences as well as *associative containers* (binary trees in disguise) provide means to iterate *linearly* over them.

The "minimal requirements" tip translates to "final coalgebras" in our framework. That aspect is unlike the approach of the Bird–Meertens formalism, which has been traditionally based on "initial algebras."

Dehnert and Stepanov [DS98] continue:

> We call the set of axioms satisfied by a data type and a set of operations on it a *concept*. Examples of concepts might be an integer data type with an addition operation satisfying the usual axioms; or a list of data objects with a first element, an iterator for traversing the list, and a test for identifying the end of the list. The critical insight which produced generic programming is that highly reusable components must be programmed assuming a minimal collection of such concepts, and that the concepts used must match as wide a variety of concrete program structures as possible. Thus, successful production of a generic component is not simply a matter of identifying the minimal requirements of an arbitrary type or algorithm — it requires identifying the common requirements of a broad collection of similar components. The final requirement is that we accomplish this without sacrificing performance relative to programming with concrete structures.

We can contrast the above to a characterization of abstract data types as classes of algebras. According to Thatcher *et al* [TWW82]:

> what is "abstract" about an abstract data type is that it consists of an isomorphism class of algebras rather than any concrete representation of the class. When it comes to specifying an abstract data type one can display a particular algebra and define the abstract data type as the isomorphism class of that algebra. The proposed alternative is to character-

ize the isomorphism class using axioms written in terms of the operations on the types.

A fundamental difference between the first school and the second school is that the latter equates linguistic support with generic programming, while the former defines it as a methodology. Furthermore, the Musser–Stepanov school promotes structuring components based on the efficiency or algorithmic complexity offered by the coalgebras, whereas those concerns appear to be secondary in the calculational approach. For example, the data structure list is usually taken as *the* canonical realization of sequences in the functional programming setting. We are not aware of work in the calculational approach, where complexity guarantees of operations (in the style of Musser–Stepanov) and genericity are given equal weight.

In a sense, the opposition of styles is similar to that of bottom-up versus top-down design. From our perspective, a good theoretical framework for generic programming should provide for mathematical tools necessary for systematic application of Dehnert and Stepanov's methodology to both the implementation and correctness proof of generic components as exhibited by the Bird–Meertens formalism.

8 Conclusions

The two approaches to generic programming, 1) as defined by the process and outcome of designing STL and similar libraries, and 2) as defined by the practice of data-type generic programming in the functional programming community, are intrinsically connected. The first approach to generic programming focuses on finding useful fundamental algebras, and defining generic functions mapping to such algebras following a final coalgebra point of view. The second, datatype generic programming, operates on initial algebras and focuses on finding algorithms on those algebras. These algorithms are applicable to a wide variety of data-types, as there are conversions from regular inductive datatypes to the structures of the functors that define them. The most interesting aspect of datatype generic programming is iteration operators for free as implementation tools and reasoning devices to capture patterns in proofs about generic functions. Combining Musser–Stepanov's methodology with a categorial approach to datatypes appears to be a promising road for systematic implementation and proof of properties about useful generic programming, and is subject for future work.

9 Acknowledgments

We are grateful to the anonymous reviewers for their comments and suggestions that improved the paper. We are grateful to Bjarne Stroustrup for suggesting the *doggiemorphism* recursion pattern, which regretfully did not fit within the space limits.

10 References

[ASS84] Hal Abelson, Jerry Sussman, and Julie Sussman. *Structure and interpretation of Computer Programs*. MIT Press, 1984.

[Bir87] Richard Bird. *Logic of Programming and Calculi of Discrete Design*, volume F.36 of *NATO AI Series*, chapter An introduction to the theory of list. Springer Verlag, 1987.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, chapter Generic Programming — An introduction, pages 28–115. Springer-Verlag, 1999.

[Coh81] Paul Cohn. *Universal Algebra*. Kluver, 1981.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DS98] James C. Dehnert and Alexander Stepanov. Fundamentals of Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11, Schloss Dagstuhl, Germany, April 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 115–134. ACM Press, 2003.

[GTWW77] J.A. Goguen, J.W. Thatcher, E.G. Wganer, and J.B. Wright. Initial Algebra Semantics and Continuous Algebra. *Journal of the Association of Computing Machinery*, 24(1):68–95, January 1977.

[Hin00] Ralf Hinze. A New Approach to Generic Functional Programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 119–132, Boston, USA, 2000. ACM Press.

[Hin04] Ralf Hinze. Generics for the masses. In *Proceedings of the ninth ACM SIGPLAN International Conference on Functional Programming*, pages 236–243, Snow Bird, UT, USA, 2004.

[Hut98] Graham Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 280–288, Baltimore, Maryland, USA, 1998.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic Programming. In *Advanced Functional Programming, Second International School-Tutorial Text*, volume 1129 of *Lecture Notes In Computer Science*, pages 68–114. Springer-Verlag, 1996.

[JJ97] Patrik Jansson and Johan Jeuring. Polyp — a polytypic programming language extensions. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Lan-*

guages, pages 470–482, Paris, France, 1997.

[JPL03] J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library : unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[Kam83] Samuel Kamin. Final Data Types and Their Specification. *ACM Transaction on Programming Languages*, 5(1):97–123, January 1983.

[McC60] John McCarty. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of The ACM*, April 1960.

[Mee86] Lambert Meertens. Algorithmics — toward programming as a mathematical activity. In J.W de Bakker and J.C van Vliet, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, North-Holland, 1986.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[ML01] Saunders Mac Lane. *Categories for the Working Mathematicians*. Springer, 2nd edition, 2001.

[MS88] David A. Musser and Alexander A. Stepanov. Generic Programming. In *In proceeding of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[R5R98] Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[RG03] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.

[Str67] Christopher Strachey. Fundamental Concepts in Programming Languages. lecture notes for the International Summer School in Computer Programming, August 1967.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[TWW82] J.W. Thatcher, E.G. Wagner, and J.B. Wright. Data Type Specification: Parameterization and the Power of Specification Techiniques. *ACM Transcation on Programming Languages and Systems (TOPLAS)*, 4(4):711–732, October 1982.