Scripting .NET using Mondrian

Erik Meijer[1], Nigel Perry[2], and Arjan van Yzendoorn[3]

[1] Microsoft
erik@microsoft.com
[2] Massey University
N.Perry@massey.ac.nz
[3] Utrecht University affie@cs.uu.nl

Abstract. We introduce the design of Mondrian, a functional scripting language for glueing together components on the .NET platform. Mondrian is monadic statement centric with pure expressions and non-strict evaluation and explores the melding of the OO and the purely lazy functional paradigms.

1 Introduction

This paper introduces Mondrian, a functional scripting language designed especially for the new Microsoft .NET platform. Mondrian is aimed at two different audiences. One is existing functional programmers who would like to be able to inter-work more closely with other languages that target the .NET Common Language Runtime (CLR), the other is existing OO programmers who would like to explore being able to write and access objects written in functional languages.

Mondrian inherits "just-in-time" (lazy) evaluation, higher order functions and monadic commands from Haskell, and classes, threads, and exceptions from the CLR (although influenced by Massey Hope+C [15] and Haskell [16]). The syntax of Mondrian is a melding of C$^\sharp$ [2] and Haskell.

The formal semantics and type rules of Mondrian are yet to be defined precisely, and indeed as an experimental language are in a state of flux. Ultimately, Mondrian's type system will depend on the availability of generics in the CLR [9]. Therefore this paper only presents a semi-formal description of the language.

The current implementation of Mondrian is highly experimental, its main goal is to explore interoperability in the .NET space, often at the expense of raw execution speed. We do however use the underlying CLR features where possible, and "encode" only features such currying and lazy evaluation that the CLR does not directly support.

2 Mondrian

The semantics of Mondrian are based of those of lazy functional programming with monadic I/O, with modifications and additions to suit an OO environment and to inter-work well with other languages that target the CLR. We describe Mondrian by highlighting the differences between it and a traditional functional language.

2.1 History and context

We started to explore the possibility of a pure functional scripting language for glueing together software components [5] in the original Mondrian paper [13]. Along the way however, we had to encountered several other interesting research problems such as foreign function interfaces [12, 10, 14, 4, 3, 8] and type system issues [18, 11] that needed to be solved before we could continue the actual implementation on Mondrian. In the mean time, Microsoft announced its new .NET platform, which made it much easier to build our experimental compiler as we could leverage on the high-level services of the Common Language Runtime (CLR) such as garbage collection, threading, language interoperability, etc.

The name *Mondrian* honors the Dutch painter Piet Mondrian (1872-1944) and reflects the purity, minimalism, orthogonality and simplicity of the language design.

The Mondrian compiler is available for download at the Mondrian web site `http://www.mondrian-script.org`. The compiler also accepts GHC's "Core" intermediate language as input, hence it can serve as a portable back end for Haskell as well.

2.2 Functions and Expressions

As a functional language, functions and pure expressions are central to the semantics of Mondrian, in contrast to object oriented languages where objects and methods are fundamental. Within Mondrian functions and expressions have a pure and lazy semantics. When viewed from another language executing on the .NET platform a Mondrian function appears as an object with a specific method `Apply` as its entry point, which conforms to the object-oriented view of the world (section 6).

Expression evaluation in Mondrian is lazy. Functions are defined using lambda notation and to simplify the language no pattern matching on arguments is provided. Function application is curried. Unusually the typical "\" used to denote the start of a lambda expression is also omitted. Top level names are bound by simple assignment syntax. For example, the addition function is defined by:

```
add = a -> b -> a + b;
```

Apart from the built-in operators, overloading (ad-hoc polymorphism) is not supported.

2.3 Types

Mondrian provides the same primitive types as the .NET platform; such as integers, floats and characters. Strings are also directly supported.

Parametric polymorphism is provided in the usual functional language manner. For example a function to evaluate the length of a list (see below) is typed as:

```
length : forall a. List<a> -> Integer
```

The CLR (and C♯) do not yet provide generics or parametric polymorphism and function types [9]. Therefore when viewed from C♯ a polymorphic Mondrian function appears as a function over `Object` (ie all type variables are erased and replaced by `Object`), hence the signature of the function `length` appears in C♯ as:

```
Integer length(List as);
```

This mapping clearly involves some static type information loss; while there can be many distinct type variables in Mondrian there is only one `Object` in .NET. Further work is required in this area to determine the most appropriate mapping and division between static and dynamic type checking.

Functional languages typically provide type products, disjoint unions and parametric polymorphism. Object-oriented languages usually are based around type products and subtype polymorphism. To bridge this gap Mondrian's type system provides products, subtypes, and parametric polymorphism.

Uses of disjoint unions in traditional functional languages are replaced by the use of classes and subclasses in Mondrian. For example, the standard list type:

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

is defined in Mondrian by defining an abstract base class `List` and two subclasses `Nil` and `Cons`:

```
abstract class List<a> {};
class Nil<a> extends List<a> {};
class Cons<a> extends List<a> {
  head : a;
  tail : List<a>;
};
```

To construct an instance of a `List` you call either `new Nil {}` to obtain an empty list, or `new Cons{ head = a; tail = as; }` to cons an element `a` on an existing list `as`. For convenience both the use of `new` and the empty field list `{}` are optional.

2.4 Multi-choice Selection and Pattern Matching

For multi-choice selection and pattern matching a switch expression is provided. The syntax is reminiscent of C♯ while the semantics comes from Haskell.

For primitive types, such as integers and characters, selection is based on the value of the predicate. For class types the selection is based on the subtype of the predicate and equates to the use `is` in C♯.

The following example, which implements the standard list map function, illustrates the `switch` expression on subtypes:

```
map : forall a,b . (a -> b) -> List<a> -> List<b>;
map = f -> as -> switch (as) {
  case Nil :
    Nil;
  case Cons { a = head; as = tail; } :
    Cons { head = f(a); tail = map(f)(as); };
};
```

2.5 Namespaces

Mondrian currently inherits the concept namespaces (and the syntax) of C♯, however, the notion of namespace does not really exist in the CLR, which instead is based on the notion of modules and assemblies. We are currently considering changing Mondrian to reflect this more directly.

2.6 Commands

Mondrian inherits monadic I/O from Haskell [21], the syntax follows Haskell except the `do` keyword is omitted in keeping with the minimal approach of Mondrian.

For example, a simple "hello world" program in Mondrian may be written as:

```
main : IO<()>;
main = {
  Console.WriteLine("Please enter you name: ");
  name <- Console.ReadLine();
  Console.WriteLine("Hello " + name);
}
```

Note: Strings in Mondrian have type `String` rather than the list of characters as in Haskell. The operator "+" is also overloaded to represent string concatenation as in C♯.

Commands are first class citizens in Mondrian, that means you can pass them as arguments to and return them from functions, put them in list, etc. In combination with lazy evaluation, this allows you to define your own control structures.

For instance, the function `forEver` executes a command forever. You can easily define `forEver` using the standard functions `repeat : a -> List<a>` that takes an element `a` and generates an infinite list of copies of `a`, and the function `sequence_ : List<IO<a>> -> IO<()>` that takes a list of commands and runs them in sequence:

```
forEver : IO<a> -> IO<()>;
forEver = ma -> sequence_ (repeat (ma));

sequence_ : List<IO<a>> -> IO<()>;
sequence_ = mas -> switch (mas) {
  case Nil:
    { return Nil; };
  case Cons{ ma = head; mas = tail; }:
    { ma; sequence_ mas; };
}

repeat : a -> List<a>;
repeat = a -> new Cons{ head = a; tail= repeat a; };
```

3 Exceptions

The execution of I/O statements within the monadic system of Mondrian has a well defined temporal order, in common with similar mechanisms in Hope+C [15] and Haskell [19]. This execution model also has direct parallels with that of the exception mechanism provided by the CLR.

3.1 Handling Exceptions

In the CLR, all exceptions are treated uniformly by replacing the current execution with the execution of an exception handler. We adopt this same model within the Mondrian monadic I/O system, and borrow most of the syntax from C♯ by adding a monadic "try/catch/finally" construct to Mondrian's commands:

```
try{
  ...
} catch(e : Exception) {
```

```
    ...
  } finally {
    ...
  }
```

For programmatically generated exceptions we also adopt the CLR model adding a keyword throw of type:

```
throw : forall a. Exception -> a
```

By adopting the same exception model as .NET we also gain inter-language exceptions. If .NET code invoked from Mondrian throws an exception then it can be caught by Mondrian code, and vice-versa.

3.2 Example

The function `readLinesFromURL` attempts to open a URL and returns the response as a list of Strings using the helper function `readLines : Stream -> IO<List<String>>`. Any failure to do this is caught in a try-catch block, which then returns an empty list of strings.

```
readLinesFromURL :: String -> IO<List<String>>;
readLinesFromURL = host -> {
  try {
    url <- WebRequest.Create(host);
    resp <- HttpWebRequest.GetResponse() url;
    str <- HttpWebRequest.GetResponseStream() resp;
    readLines str;
  } catch (e : Exception) {
    result Nil;
  };
};
```

4 Concurrency

Mondrian supports concurrent programming using threads. So it can inter-work closely with other .NET languages it builds directly on the threads and synchronization primitives of these environments. In particular:

- A Mondrian thread is a (subclass of a) .NET thread. This not only provides an obvious implementation solution but also trivially supports a mix of Mondrian and .NET threads inter-working with each other.
- Mondrian monadic commands can be enclosed within a monitor (synchronized block), and have access to the same synchronization primitives (wait, notify/pulse).

– Mondrian has access to the same IPC mechanisms as other .NET languages. This includes monitors as above and also standard classes for pipes, object streams, etc. Using these mechanisms IPC works seamlessly between Mondrian threads or between a Mondrian thread and one written in another .NET language.

The first two points are supported by additions to Mondrian, the last is already provided through the Mondrian to foreign language calling interface.

4.1 Threads in Mondrian

Mondrian provides its own `MondrianThread` class which parallels the one provided by the CLR. To match the function orientation of Mondrian the `Thread` class provides a number of functions whose names and purpose follow those of the CLR, but which take a `MondrianThread` object as argument:

```
CreateThread : forall a. IO<a> -> IO<MondrianThread<a>>
```

This creates a thread which executes the given Mondrian command and returns a Mondrian thread object. The returned object is a subtype of the .NET `Thread` and contains all the standard methods.

As in .NET once created a thread must be started using the function:

```
Start : Thread<a> -> IO<a>
```

Note this function takes a `Thread` and not a `MondrianThread`, so it can start either a Mondrian or foreign language thread. Standard methods on threads such as `join` and `suspend` are similarly provided.

The following simple example creates two threads that write an infinite stream of `"a"` respectively `"b"`'s on the screen, and then runs them in parallel:

```
main = {
  as <- CreateThread { forEver(Console.Write("a")); };
  bs <- CreateThread { forEver(Console.Write("b")); };
  Start(as); Start(bs);
}
```

4.2 Inter-Thread Synchronization and Communication

Mondrian provides a direct equivalent to C♯'s synchronized blocks:

```
synchronized (e){ ...statements... }
```

with type:

```
forall a, b. Lock -> IO<b> -> IO<b>
```

and the associated functions:

```
getLock : IO<Lock>
wait : Lock -> IO<()>
notify : Lock -> IO<()>
notifyAll : Lock -> IO<()>
```

To preserve Mondrian's functional semantics, we cannot lock on an arbitrary object, but have to obtain a lock explicitly via the monadic call `getLock`. The execution of a synchronized statement first evaluates the expression `e` to WHNF to obtain the object to lock and the statements are executed. On completion the lock on the object is released. The semantics of the functions `wait`, `notify` and `notifyAll` directly mirror their .NET counterparts.

4.3 Other IPC Methods

Mondrian also provides a library of constructs ranging from the low-level semaphore through typed channels constructed over .NET pipes so that distributed systems can easily be constructed. Even Haskell's MVars [7] are provided. We stress though that all these are simple constructions provided as a library and involve no extensions to the Mondrian language or its primitive operations.

5 Implementation

Implementing a functional language such as Mondrian on the .NET platform requires a number of issues to be tackled, the main ones being:

- How are values and parametric polymorphism represented?
- How are functions are represented?
- How is partial function application (currying) handled?
- How are "just in time" computations (thunks) represented?

All these issues are well understood and methods have been developed for efficient implementation on standard machine architectures. However the CLR differs from traditional machines in a number of ways, including:

- It is a typed machine. The type system for any language implemented on .NET has to map to the .NET type system.
- It is object-oriented, the key building blocks are objects with methods. Mondrian is function-oriented, it's key building blocks are functions. Currently, the CLR does not yet natively support closures.

We choose to implement Mondrian by using features of the .NET platform wherever possible and resorting to well-known techniques for implementing functional languages, the Spineless Tagless G-Machine (STG)[6] or the so-called *PUSH/ENTER* model, in case there is no direct support for a semantic feature in the CLR.

5.1 The Representation of Values and Functions

All values in Mondrian are first class and exist as long as required (referenced). The standard method of implementing such values in any language is to use a garbage collected heap. The object system of .NET is heap based and automatically managed by the system. The obvious mapping is therefore taken, a Mondrian value is represented as a .NET object.

Mondrian, in common with other functional languages, is built around functions. Functions can be defined statically in the program; functions are first class values and can be passed as arguments, returned as results, and stored in data structures; and functions can be created at runtime based on other dynamic values.

In .NET an object possess all the properties of a function in Mondrian. We therefore map a Mondrian function to a .NET object that implement the `Code` interface that contains a method `ENTER` that is used to evaluate the function by the Mondrian runtime:

```
interface Code {
  public Object ENTER();
}
```

For example, given the Mondrian function definition:

```
Foo = x -> ...
```

is represented by the following .NET class[1] `Foo` that implements the `Code` interface:

```
class Foo : Code {
  public Object ENTER() {
    ... code for body of Foo ...
  }
}
```

To handle nested functions, we simply create an object with private fields to store any values that are needed to construct the function (a *closure*), and compile code for the method which performs the computation using these values. A

[1] We show our target code in C♯ for readability

closure also implements the `Code` interface and can be used just like a compile-time defined function.

The design space of representing function closures in an object oriented framework is a surprisingly large, and we describe only a tiny slice of that here, in particular we will not discuss the duel *EVAL/APPLY* method that has been investigated by other functional languages that target the CLR such as ML.NET [1]. We also are looking in representing closures using delegates.

5.2 Function Arguments and Partial Applications

A powerful feature of Mondrian, in common with many functional languages, is the ability to "partially apply" (or curry) a function, that is call a function with fewer arguments than it is defined to accept. If fewer arguments are supplied the result is a partially applied function, which is just a dynamically generated function with some argument values already wired in. These partial applications can therefore be represented just like any other function.

However a method is needed to actually pass the arguments to a function in the first place. As .NET is strongly typed simply missing out some of the arguments and then calling the function will not work. In other words, the CLR does not directly support curried functions, and hence we have to implement this semantic feature ourselves. One possible solution is to create a separate stack object to pass function arguments. A function call then becomes: place arguments on stack using `Mondrian.PUSH`, call the `ENTER` method of the function's object passing no arguments (an obvious optimization is to have several overloaded `ENTER` methods that serve as fast entry points when a known function is called with a known number of arguments).

For example, adding more detail to the `Foo` function given above its definition becomes:

```
class Foo : Code {
  public Object ENTER() {
    Object arg = Mondrian.POP();
    ... rest of body of Foo ...
   }
}
```

A call to this function will be compiled to C♯ code which is equivalent to:

```
Foo f = new Foo();
Mondrian.PUSH(<argument>);
result = f.ENTER();
```

5.3 "Just In Time" Computations

Mondrian is a lazy language and does not actually perform a computation until its result is needed, it does computation "just in time". This means any value may in fact be either a real value or a computation which when performed will produce the real value (a so called *thunk*). The essence of lazy evaluation is to update thunks with their values after they have been evaluated for the first time. Like currying, lazy evaluation is not a semantic feature that is directly supported by the CLR, so we have to implement it explicitly.

Again, the design space for implementation thunks is surprisingly big. We have experimented with various implementations, and have not yet decided which one is best. The one sketched below using exceptions in an interesting way.

The standard STG implementation method solves this problem by wrapping JIT computations using a helper function which pushes a special value on a "mark stack" which encodes the current stack depth and a reference to the thunk object being evaluated. Now on every entry to a function a check is made on the number of arguments available and the value on top of the mark stack. When either the argument stack is empty (computation complete), or there are insufficient arguments (JIT computation has produced a partial application), the value on the mark stack can be removed, the thunk updated to hold its value, and then the process repeated on the new top of the mark stack.

All this may sound a bit complicated and indeed it is! However using the facilities of .NET we developed a simple solution for lazy evaluation. Rather than wrap a JIT computation in a small function which pushes a value onto a mark stack, we use one which does a function call and starts up a new *trampoline*, thereby effectively using the runtime stack instead of maintaining an explicit marker stack. A trampoline is a small iterative loop, that makes repeated calls to a function until the value returned indicates the computation has completed.

As explained the mark stack method iterates down the mark stack updating as many thunks as needed and handles the case of partial applications. If it did not then the trampoline would have to distinguish between three possible return types; a simple value, a continuing computation, and a partial computation. This is where we can now use the features of .NET to simplify the implementation. We wrap the trampoline in an exception (try/catch) block and then on function entry throw an exception, carrying a partial application value, if there are insufficient arguments. The trampoline/exception block catches this and updates the JIT object. The exception is then re-thrown, which has the same effect as the iterative compare/update/pop cycle of the mark stack method.

The resultant code is a lot simpler than the mark stack method, and does not require a special stack and the overhead of maintaining it.

A sketch of this code in C♯ is as follows, if a function does not find enough arguments on the argument stack, it throws and *partial application exception*:

```
class Foo : Code {
```

```
    public Object ENTER()
     {  if(!Mondrian.AVAILABLE(1)) throw PAP_Exception(this);
        Object a = Mondrian.POP();
        <body of Foo>
     }
}
```

And the JIT trampoline follows the following schema:

```
try {
  // Trampoline loop
  while(f is Code) { f = f.ENTER(); }
} catch(PAP_exception e) {  /
  ... update thunk with ...
  ... partial application f ...
  throw e;
}
```

Thunks and Exceptions It is possible for the execution of I/O to cause the evaluation of a thunk. If an exception occurs during the evaluation of the thunk then the thunk must be left in a suitable state. This issue can be a thorny one to deal with [17], however due to Mondrian being based on the CLR model which does not support asynchronous exceptions the problem is easily handled. Within the thunk evaluation code we catch any exception, update the thunk with a closure which throws the same exception value, and then re-throw the exception.

Thunks and Threads Once multiple evaluation threads are running there is the possibility that two, or more, threads may attempt to evaluate a thunk. As the evaluation of a thunk can take an arbitrarily large amount of time it is best to avoid multiple evaluation. To handle this we make the evaluation method for a thunk a synchronised one. The first thread to attempt to evaluate the thunk will acquire the lock and undertake the work. Any subsequent threads will block until the lock is released and then find the work done and use the already computed result.

6 Interop

6.1 Calling C♯ from Mondrian

C♯, or any other .NET language, are invoked from Mondrian through the I/O system to preserve Mondrian's purely functional semantics. To call a constructor

the `create` construct is used which returns a monadic function closure which when executed invokes the foreign language constructor to create an object.

To enable the calling of a foreign method, the `invoke` (and `invokeStatic`) construct is provided. This returns a monadic function closure which takes an object along with any method arguments and calls the foreign method. This style of method calling where the object is passed explicitly differs from the C♯ model but is reminiscent of the Ada 95 OO model [20].

The function closures returned by `create` and `invoke` evaluate any argument values to WHNF before invoking the foreign constructor/method. This is done to bridge the gap between the lazy world of Mondrian and the strict world of .NET. However should a structured value need to be passed then either it must be fully evaluated before the inter-language call (using Mondrian's strict function, see below), or the foreign function must call back to Mondrian to evaluate any suspensions it finds.

The following simple example demonstrates `create` and `invoke`:

```
main = {
  gen <- create System.Random();
  n <- invoke System.Random.Next(Int32) 10 gen;
  putStrLn ("Random 1..10: " + show n);
}
```

Note that `create` and `invoke` take both the foreign constructor/method and it's argument type(s), if any. The Mondrian type system does not support overloading so these constructs use the argument types to uniquely identify the constructor/method to be called.

Accessing the fields and properties of a foreign object is provided by the `get` and `set` constructs in a similar manner as for constructors and methods.

6.2 Calling Mondrian From C♯ And Other .NET Languages

Programmers calling Mondrian from other .NET languages should not need to know how Mondrian works under the hood. In particular, partial application is not normally used by other .NET languages and when they need to call Mondrian code to perform some algorithm, they will usually therefore supply a full set of arguments to the Mondrian function. To handle this common scenario, we simply add a `Apply` method with the full number of arguments to the function object, which is then used by other .NET languages as the function's entry point.

For example, our function `Foo`, with the method added for calling from other .NET languages it now looks something like:

```
class Foo : Code {
  // used by Mondrian runtime
```

```
    public Object ENTER() {
      ... as before ...
    }

    // used when called externally .NET languages
    public Object Apply(Object x) {
       Code f = this;
       ...evaluate f in Mondrian specific way...
       return f;
    }
}
```

And a call from C♯ becomes something like:

```
Foo foo = new Foo();
result = foo.Apply(4);
```

Note that `Apply` is not a static method on `Foo` as evaluating `foo.Apply(3)` may trigger thunk updates. Indeed calling `foo` a subsequent time might return its result faster than calling `foo` the first time.

References

1. The SML.NET Compiler. http://research.microsoft.com/Projects/SML.NET/.
2. Ben Albahari, Peter Drayton, and Brad Merrill. *C♯ Essentials*. O'Reilly, 2001.
3. Sigbjorn Finne, Erik Meijer, Daan Leijen, and Simon Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *Proceedings of ICFP'99*.
4. Sigbjorn Finne, Erik Meijer, Daan Leijen, and Simon Peyton Jones. HDirect: A Binary Foreign Function Iinterface for Haskell. In *Proceedings of ICFP'98*.
5. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
6. Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2), April 1992.
7. Simon Peyton Jones, Andy Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL*, 1996.
8. Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of ICSR5*, 1998.
9. Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI*, 2001.
10. Daan Leijen, Erik Meijer, and Jim Hook. *Haskell as an Automation Controller*, volume 1608 of *LNCS*. 1999.
11. Jeff Lewis, Mark Shields, Erik Meijer, and John Launchbury. Implicit Arguments: Dynamic Scoping with Static Types. In *Proceedings of POPL'00*.
12. Erik Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, 2000.
13. Erik Meijer and Koen Claessen. The Design and Implementation of Mondrian. In *Haskell Workshop*, 1997.

14. Erik Meijer, Daan Leijen, and Jim Hook. Client-side Web Scripting with HaskellScript. In *PADL*, 1999.
15. Nigel Perry. *Massey Hope+C*, 1992. Massey University.
16. Simon Peyton-Jones, John Hughes, and (eds). Report on the Language Haskell'98. http://www.haskell.org/report, February 1998.
17. Alastair Reid. Putting the Spine back in the Spineless Tagless G-Machine: An Implementation of Resumable Blackholes. In *IFL*, 1998.
18. Mark Shields and Erik Meijer. Type Iindexed Records. In *Proceedings of POPL'01.*
19. Andy Moran Simon Marlow, Simon Peyton Jones and John Reppy. Asynchronous Exceptions in Haskell. In *PLDI*, 2001.
20. S. Tucker Taft and Robert A. Duff, editors. *ADA 95 Language and Standard Libraries : ISO/IEC 8652:1995(E)*. Springer Verlag, 1997.
21. Philip Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, 1995.