Implementing Functional Languages on Object-Oriented Virtual Machines

Nigel Perry[1,2]

New Zealand

Erik Meijer

Microsoft Corporation

Abstract

We discuss some of the issues involved in implementing functional languages on object-oriented virtual environments (VE) such as the Java Virtual Machine and Microsoft's .NET. The paper concentrates on how functional language type systems can be supported in these OO-typed environments, and on how functional and OO languages can interwork. Mondrian, a new functional language specifically designed for such environments, is introduced as an example.

1 Introduction

Over recent years the idea of a using a common virtual machines/binary formats to support multiple languages and/or ease portablity has resurfaced. It is not a new idea; examples include the Fortran, Algol and Cobol compilers on ICL 1900 machines in the 70's and the Amsterdam compiler kit [1]. The major differences today are that the VEs are strongly typed, have object-oriented (OO) type systems, and have gained widespread adoption.

The most common of these VEs at present is the Java Virtual Machine (JVM) [2] developed by Sun and available on most major computer platforms. The JVM has been closely tied with the Java language. Other languages targeting the JVM are not in widespread use, but examples do exist such as SML-J [3] and Component Pascal [4].

Recently Microsoft has announced its .NET project which will be a core component of future versions of Windows, and may also be implemented on other OSes. This VE has been designed to support multiple languages from the outset, with Microsoft themselves supplying C#, C++ and Visual Basic.

The widespread availability of these VEs make them an important target for language designers. In this paper we discuss the implementation of functional languages on these platforms. Due to the rather different type systems and evaluation models used by functional languages there is a gap to bridge when implementing them on OO VEs which does not exist when compiling directly to untyped assembly language.

We will not argue the merits of functional or object-oriented languages, there is plenty already written on that subject, but rather work from the basis that both have their merits. We will also not explore multi-paradigm languages, such as O'CAML [5], we see the benefit of

[1] This work was started while both authors were at Utrecht Universiteit, the first on sabbatical from Massey University. Many thanks also go to Arjan van IJzendoorn who was an original member of the team at Utrecht. The work was supported with grants from the Dutch Government and Microsoft Corporation.

[2] NPerry@mac.com

targeting a common VE as allowing multi-paradigm programming without the need to design multi-paradigm languages. Each may have its place, but we leave that for others.

1.1 Benefits

The advantages of targeting these VEs are many and include portablity, access to large libraries, inter-language working opportunities, and garbage collection.

By targeting a standard VE we immediately gain access to all the computing platforms it runs on, a major advantage for compiler writers and their language users. The VEs also come with large standard libraries (frameworks) for such common tasks as GUI's and networking. Being able to access these is a major benefit to programmers and language providers alike. Along with this is the ability to easily access code written in other languages easing the combination of appropriate tools to produce an overall solution.

Finally, for languages which require garbage collection, which includes functional languages, the VEs provision of this service removes a major source of obscure errors that often take many hours of debugging to track down!

1.2 Issues

However targeting these standard VEs raises some problems for functional language implementors. The type systems of the two paradigms are quite different. The VEs support a system based on sub-typing (inheritance), overriding ("polymorphism") and immediate ("strict", "applicative order") evaluation. Functional languages are typically based on parametric polymorphism and just-in-time ("non strict", "normal order") evaluation.

In this paper we examine bridging this divide. We use Haskell [6] as one of our example functional languages. We also briefly introduce a new functional language, Mondrian [7], which has been designed as a result of this work to enable easier inter-working with OO languages. Mondrian itself is still in development and its final syntax and semantics may change. We restrict ourselves primarily to type and inter-language working issues. We also restrict ourselves to the intersection of the JVM and .NET VEs and do not explore their unique features.

2 Mapping Types

2.1 Primitive Types

Supporting primitive types is trivial as they are common to both models. The VEs provide both value ("unboxed") and reference ("object", "boxed") versions of primitive types. To easily support parametric polymorphism functional language implementations often restrict themselves to reference versions of primitive types, and this is the approach we have taken with Mondrian.

The VEs value types can be used by functional language compilers to produce optimised code under certain circumstances, but this is usually transparent to the user. The Glasgow Haskell Compiler [8] provides extensions to Haskell to directly support unboxed primitive types and expose them to the programmer.

2.2 Type Products

The product, or tuple, type is widely used in functional languages. For example a type for a coordinate might be expressed in Haskell as:

```
type Coord = (Float, Float)
```

or by a record declaration such as:

```
data Coord = Cd { x, y :: Float }
```

The direct equivalent of both of these is a *field only* class:

```
class Coord { Float x; Float y; }
```

A Haskell compiler can use this translation. In Mondrian we have chosen to make the representation explicit and use the class construct directly.

Of course not all classes supported by the VEs are field-only, or else there would be no methods! The above mapping only allows a class in the VE to be viewed as a product by a functional language. To access the methods of a VE class from a functional language requires a different mechanism, this is covered later under inter-language working.

2.3 Type Unions

Tagged unions are central to functional languages. They provide for both simple enumeration types and more complex variant and recursive types. The key role of tagged unions is reflected in the provision of pattern matching to provide simple and concise access to their components.

For example in Haskell a list of integers may be expressed as:

```
data IntList = Nil | Cons Integer IntList
```

Such union types are not provided by the VE type systems, OO languages replace type union by subtyping. We can map a union type onto this model by defining an abstract class to represent the type and a subclass for each of its variants:

```
abstract class IntList {};

class Nil extends IntList {};

class Cons extends IntList
{  head : Integer;
   tail: IntList;
};
```

These two are *not* equivalent, the former supports a single level of variants and the number of variants is fixed. In the latter it is possible both to subclass the variants and to extend the number. This means that not all type hierarchies of field-only classes on the VE can be viewed as a tagged union by functional languages such as Haskell. For this reason Mondrian uses the latter formulation directly, giving it a more flexible type system in this area than traditional functional languages.

When functional languages are compiled union types are normally implemented using a record structure with a tag to indicate the variant. Pattern matching is compiled into code

which tests these tags. Subtyping is typically implemented the same way, with subtypes being distinguished by a tag. OO languages directly support the determination of which subtype a particular reference is by providing an operation, for example “instanceof”, which tests the tag. This operation is provided as an instruction in the VEs. Thus, though the union and subtype formulations look quite different, they are typically implemented in exactly the same way, so there is no hidden cost in the above mapping.

2.4 Parametric Types

Our list example would not usually be written as above in a functional language, instead a parametric type would be used which represents lists of any type:

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

This type represents an *homogenous* list in Haskell. All members of the list will be the same type, and this is checkable and enforced at compile time.

Neither the JVM or .NET VEs directly support such parametric types, though proposals have been made to include them in the JVM [9]. However both VEs support a rooted type hierarchy with all (reference) types deriving ultimately from “Object”. This type model supports the standard OO model of *heterogeneous* generic types where, for example, a list might be defined as:

```
class ListNode
{  Object head;
   ListNode tail;
};
```

This formulation suggests the way to compile parametric types: replace type parameters with Object. For example, our Haskell list becomes:

```
abstract class List {};

class Nil extends List {};

class Cons extends List
{  head : Object;
   tail: List;
};
```

Again for Mondrian we directly support a more OO style with the addition of type parameters:

```
abstract class List<a> {};

class Nil<a> extends List<a> {};

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

Unlike the other mappings the strategy for parametric types carries a cost. The VEs are strongly typed. When an item of type Object is used in a context which requires a particular type then a runtime type check must be performed. The original type cannot in general be determined statically. This is an accepted cost in OO programming. For example when the *head* field is accessed in the `ListNode` class the OO programmer must supply a type cast:

```
Integer anInt;
ListNode node;

node.head = anInt; // OK
anInt = (Integer)node.head; // cast required
```

With parametric types the type of any value, or at least that two types are compatible, can be determined statically, and no runtime checks are required. In our mapping onto the VEs runtime type checks will occur, which will always succeed. There is currently no trivial way to avoid this, direct support for generic types in the VEs would address the issue. We have no figures to quantify this cost at present, however the cost is clearly no greater than that normal for OO languages.

3 Functions

In OO languages functions are members of classes while functional languages usually follow the traditional module & function model. This difference has a fundamental effect on the styles of programming in the two paradigms. For example, in a functional language, in common with many imperative languages, a sorting function may take a comparison function as an argument. In the OO equivalent an object with a well known method would be used.

In implementation terms this means that while an object is a value which can be manipulated a function is not. However the ability to manipulate function values is essential for functional language implementations.

3.1 Representing Monomorphic Functions

As suggested by the sorting example above, a function in a functional language could be compiled into an class with a well known method. A function "value" would then become an object reference which can be handled uniformly like any other reference type. This includes passing functions as arguments, returning them from functions and storing them in data structures.

For example, the Haskell function:

```
length :: IntList -> Integer
```

can be compiled to:

```
class length
{  Integer Eval(IntList l) { … }
}
```

The .NET VE directly supports object + method pairs with its *delegates.* We have been carefully to produce a compilation strategy which works for both the JVM and .NET so we have chosen not to exploit delegates. However it is currently uncertain whether doing so would improve the compilation strategy significantly, though they may enhance .NET's user-level programming model.

3.2 Polymorphic Functions

Polymorphic functions can be compiled using a combination of the strategies for parametric types and monomorphic functions above. For example, the Mondrian function:

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

is also compiled to:

```
class length
{  Integer Eval(IntList l) { … }
}
```

This strategy carries the same potential for run time type checks, all of which will succeed, as that for parametric types.

3.3 Partial Applications

Many functional languages allow a function to be partially applied (or *curried*); that is a function may be applied to fewer arguments than it requires and the result is a function value which expects the remaining arguments.

A typical implementation of partial applications is to construct a “closure” or “thunk”. This is simply a record containing the values of the already supplied arguments, termed the *environment*, and a reference to the original function. Following our strategy above the environment can be stored as instance variables within the class.

For example, the Mondrian code fragment:

```
times : Float -> FLoat -> Float;
…
addTax = times 1.125;
```

can be compiled into the pseudo code (based on C#):

```
class partialTimes
{  Float a1;

   partialTimes(Float a1) { this.a1 = a1; }

   Float Eval(Float a2) { return times.Eval(a1, a2); }
}
…
gst = new partialTimes(1.125);
```

3.4 Just-In-Time Evaluation

JIT evaluation[3], usually termed non-strict evaluation in functional languages, involves delaying the evaluation of an expression until its value is actually needed. A typical functional language implementation provides these in exactly the same way partial applications, except in this case there are no pending arguments. Therefore we use exactly the same compilation strategy for the VE. For example in pseudo code a JIT call to `times` is represented by:

---

[3] Mondrian’s contribution to buzz-speak! First coined by Erik.

```
class JIT_Times
{  Float a1, a2;

   partialTimes(Float a1, Float a2)
   {  this.a1 = a1;
      this.a2 = a2;
   }

   Float Eval() { return times.Eval(a1, a2); }
}
```

This compilation strategy results in a JIT "value" being just a reference. Under our compilation strategy all types are represented by reference types, so all values can trivially either be a reference to a "real" value[4] or to a JIT class instance. To support this all instance variables are declared as type `Object`. For example, the `Cons` variant of our Haskell list of integers:

```
data IntList = Nil | Cons Integer IntList
```

is actually compiled as:

```
class Cons extends IntList
{  head : Object;
   tail: IntList;
};
```

just like its parametric List equivalent.

This strategy raises the issue of efficiency. A naive implementation may contain a lot of redundant testing for JIT references as every load from an instance variable could potentially require such a check. Fortunately standard strictness analysis techniques developed for conventional functional language implementations are applicable and many of the redundant checks can be removed.

4 Inter-language Calling

The compilation strategy described above enables functional languages to be executed on the VE platforms. Despite some of the efficiency issues mentioned above it works well, even in a naive compiler. Our Mondrian system, which has been developed to initially support scripting [7], is rather naive and does little optimisation. However the performance of the resulting code is somewhere between that of Hugs [10] and GHC [8], which is more than acceptable for a scripting language.

However being able to compile to the VE only realises some of the potential benefit; portablity and "free" garbage collection. To fully realise the potential the compiled functional language code needs to be able to call, and be called from, other language code running on the VE. This includes access to the standard frameworks. Three issues need to be addressed: the handling of JIT references, calling from functional to OO code, and calling from OO to functional code.

[4] Technically a reference to a value in WHNF.

4.1 The JIT Problem

Supporting JIT evaluation for the functional language, as described above, raises a problem for inter-language working. Every access to an instance variable of a functional language constructed object requires testing for a JIT reference. Fortunately object-oriented programming style comes to the rescue. It is bad practice to allow direct access to instance variables of classes, rather access method are used. Our compilation strategy follows this and the access methods, which are just user defined functions in the source functional language, handle the testing for, and evaluation of, any JIT references.

4.2 Calling OO Code From Functional Code

To the functional language designer this issue equates to handling I/O, an area which has been well researched. Successful techniques have been developed over the years, including continuations [11] and monads [12], the latter having now been adopted as standard in Haskell. To a functional language calling a procedure in an imperative, or object-oriented, language is just the same as performing an I/O operation [13].

We shall use Mondrian to demonstrate how inter-language calling can be supported. Following on from Haskell, Mondrian uses monadic I/O. For example, the standard “Hello” program is written as:

```
main : IO Void;

main =
{  PutStr("Please enter you name: ");
   name <- GetStr();
   PutStr("Hello there " + name);
}
```

To support calling other languages operating in the VE we introduce two constructs:

- `create`, which takes the specification of a class constructor and produces a monadic version of it; and
- `invoke`, which does the same for methods.

This is easiest explained by example, the following Mondrian program runs on .NET and produces a single random number by accessing the .NET framework:

```
// produce a monadic constructor for System.Random
dotnetRand : IO System.Random;
dotnetRand = create System.Random();

// produce a monadic interface to the Random.Next method
nextRand : Integer -> System.Random -> IO Integer;
nextRand = invoke System.Random.Next(Int32)

main =
{  gen <- dotNetRand;
   num <- nextRand 10 gen;
   putStrLn ("Random 1..10: " + show num);
}
```

This approach fits seamlessly into the monadic system giving functional language programmers access to other languages using a model they are familiar with.

4.3 Calling Functional Code From OO Languages

Calling functional code from OO languages is not as seamless as the other way around. This is because traditional functional languages have no concept of “object” and “method”, and certainly not of the state an object typically embodies. This means that without making additions to the functional languages they are unable to present an OO “view” of themselves to OO clients.

For example an OO client would expect an abstract data type (ADT) to be represented by a class with instance methods for manipulating its values. However under our compilation strategy, which follows the functional data model, an ADT is represented by a *data-only* class and a *set* of method-classes.

Despite this mismatch of models calling functional code from an OO language is not difficult, and requires no changes to either language. We shall demonstrate this by example.

The following Mondrian code produces a non-strict list of primes, a task which is quite a bit harder to code in a strict language such as Java:

```
// remove all multiples from a list
sieve : List -> List;
sieve = xs ->
   switch (xs)
   {  case (y::ys):
         y :: sieve (filter (notMultiple y) ys);
   };

notMultiple : Integer -> Integer -> Boolean;
notMultiple = x -> y -> not (y % x == 0);

// the “infinite” list [n..]
from : Integer -> List Integer;
from = n -> n :: from (n + 1);

// the “infinite” list of primes
primes : List Integer;
primes = sieve (from 2);
```

The following Java code uses the above to display *count* primes starting from the *N*th:

```
void PrintPrimes(int N, int count) throws Exception
{  // list of all the primes...
   Object longList = (new primes()).Eval();

   // list access objects - from Mondrian
   tl myTail = new tl();
   hd myHead = new hd();

   // skip to first desire prime
   while(--N > 0)
   {  longList = myTail.Eval(longList);
   }
```

```
        // display primes
        while(count-- > 0)
        {  Console.out.println( myHead.Eval(longList) );
           longList = myTail.Eval(longList);
        }
    }
```

As can be seen the Java code retains its imperative style. However a Java prime generator would probably be written in “iterator” style as a class with a method to advance to the next prime.

If desired this latter style can easily be provided by writing it in the OO language. For example in Java:

```
public class Primes
{  Object longList;
   tl myTail;
   hd myHead;

   public Primes()
   {  longList = (new primes()).Eval();
      myTail = new tl();
      myHead = new hd();
   }

   public Integer Current()
   {  return (Integer)myHead.Eval(longList);
   }

   public void Next()
   {  longList = myTail.Eval(longList);
   }
}
```

Future work will examine the specification of such interface classes directly in Mondrian.

5 Conclusions

We have shown a compilation strategy for supporting non-strict functional languages on OO virtual environments such as JVM and .NET. Though there are clear mismatches between the functional and object-oriented computational models the compilation strategy is straightforward, even for the JVM which was not specifically designed to support multi-paradigm programming.

The resultant code does suffer from some inefficiencies, many of which could be addressed by supporting generic classes in the VEs. Generic classes would also benefit other language paradigms and indeed have been suggested for Java.

Our experimental language Mondrian is available for free download [14]. Also available is a demonstration version of the Glasgow Haskell Compiler modified to use Mondrian as a compilation route for .NET.

In our future work we hope to explore the design space for extending functional languages to provide OO-style access to functional code. However at this stage we have no plans to explore the integration of OO and functional approaches into a single language. Rather we plan to explore multi-language mutli-paradigm programming on the VE platforms as an alternative to designing multi-paradigm languages.

6 References

[1] A. S. Tanenbaum, H. v. Staveren, E. G. Keizer, and J. W. Stevenson, "A Practical Toolkit for Making Portable Compilers," *CACM*, vol. 26, pp. 654-660, 1983.

[2] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification*, 2 ed: Addison-Wesley, 1999.

[3] N. Benton and A. Kennedy, "Interlanguage Working Without Tears: Blending SML with Java," presented at International Conference on Functional Programming, 1999.

[4] K. J. Gough and D. Corney, "Implementing Languages Other than Java on the Java Virtual Machine," presented at Evolve 2000, Sydney, 2000.

[5] D. Rémy and J. Vouillon, "Objective ML: An effective object-oriented extension to ML," *Theory And Practice of Objects Systems*, vol. 4, pp. 27-50, 1988.

[6] S. P. Jones, J. Huges, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, "Report on the Programming Language Haskell 98," 1999.

[7] E. Meijer, N. Perry, and A. v. IJzendoorn, "Scripting .NET using Mondrian," presented at ECOOP, 2001.

[8] Glasgow Haskell Compiler: http://www.haskell.org/ghc/.

[9] G. Bracha, "JSR #000014: Add Generic Types to the Java™ Programming Language," Sun Microsystems, 1999.

[10] Hugs 98: http://www.haskell.org/hugs/.

[11] N. Perry, "Functional Language I/O," Imperial College, Unversity of London, FPG Report IC/FPR/LANG/2.5.1/29, July 88. 1988.

[12] P. Wadler, "Comprehending Monads," presented at LFP, Nice, 1990.

[13] N. Perry, "I/O and Inter-language Calling for Functional Languages," presented at Proceedings IX International Conference of the Chilean Computer Society and XV Latin American Conference on Informatics, Chile, 1989.

[14] Mondrian: http://www.mondrian-script.org.