Compiling Non-strict Functional Languages for the .NET Platform

Monique Monteiro
(Center of Informatics, Federal University of Pernambuco, Brazil
mlbm@cin.ufpe.br)

Mauro Araújo
(Center of Informatics, Federal University of Pernambuco, Brazil
mscla@cin.ufpe.br)

Rafael Borges
(Center of Informatics, Federal University of Pernambuco, Brazil
rmb2@cin.ufpe.br)

André Santos
(Center of Informatics, Federal University of Pernambuco, Brazil
alms@cin.ufpe.br)

Abstract: In this work, we propose a compilation strategy for non-strict functional languages targeting the Microsoft .NET Platform, a multilanguage platform which provides a large number of services to aid current software development. This strategy is based on the push/enter execution model, enables fast function calling mechanisms whenever possible and males use of new features present in .NET Framework, such as delegates and tail calls. Our case study was the compilation of the Haskell language, a standardized and well known non-strict functional language. Our main contribution is the construction of an environment for the testing of different compilation techniques for functional languages targeting .NET.

Key Words: .NET, languages interoperability, virtual machines, functional programming, compilers, Haskell.
Category: D.3.3

1 Introduction

The main motivation for the use of functional languages comes from features like absence of side-effects, higher-order functions and non-strict evaluation, which enable higher productivity and a more adequate framework for formal correctness proofs. However, the absence of rich APIs and productive development platforms makes difficult the adoption of these languages in large scale projects. On the other hand, there is already a rich set of tools and APIs for object-oriented languages, including virtual machines coupled with complete development platforms that offer portability and services to help developers better support applications' infrastructure requirements.

In this context, we have the opportunity to reuse such infrastructure for the functional paradigm by means of compilation techniques capable of translating programs written in functional languages to programs written for the platforms supported by those OO environments. Another possibility is the use of "bridges" to convert the calls between the two worlds, which has the disadvantage of not fully exploiting the benefits of the platform (i.e.: versioning mechanisms, code security and garbage collection – two execution environments require the existence of two garbage collectors, which incurs a huge overhead). Bridges have other disadvantages, such as the poor performance generated by the marshaling of data and the use of reflection APIs. Finally, the OO environment would have to access unmanaged code in order to communicate with the functional side.

In this work, we propose a compilation strategy for non-strict functional languages targeting the Microsoft .NET Platform. The strategy is based on the push/enter execution model [Marlow and Jones, 2004]. Our approach has two advantages over other non-strict functional languages implementations for .NET: it avoids the excessive generation of classes and enables a fast function calling mechanism wherever possible. Our case study was the compilation of Haskell [Haskell, 2002], which was chosen because it is purely functional, standardized and widely used by the non-strict functional programming community.

This paper is organized as follows. Section 2 gives a brief introduction to the .NET Platform. An overview of possible compilation strategies for compiling functional languages to object-oriented virtual machines is given in Section 3. In Section 4 we present the Haskell .NET compiler. The conclusions and future works are discussed in Section 5 and we cite related works in Section 6. Appendix A presents the compilation rules used in our solution.

2 Microsoft .NET Platform

Microsoft .NET Platform is a multilanguage development platform composed by a large set of libraries – for database manipulation, XML processing, graphical user interface components, web development, etc. – and an execution environment – the Common Language Runtime [Box and Sells, 2003], or CLR for short.

The Common Language Runtime may be defined as a stack-oriented virtual machine responsible for translating code written in the Common Intermediate Language [ECMA, 2002] – CIL – to processor-specific instructions. This translation is carried out on demand by means of *Just In Time* (JIT) compilation. So, when we say "to compile language X for the .NET Platform", we mean "to build an X compiler which is capable of generating CIL code". By stack-oriented we mean that the instructions for the machine can push operands on an evaluation stack, operate on the top of stack operands and pop operands off the stack and store them in memory or in local variables.

The CLR has several other functions such as code security, memory and threads management, with no dependencies on any specific language.

Another example of virtual machine is the JAVA Virtual Machine (JVM) [Lindholm and Frank, 1999], which has been already targeted by some functional languages implementations. However, the JVM has the following disadvantageous factors for the support of functional languages:

- it is intended mainly for supporting the JAVA language [Gosling et al., 2000], therefore targeting and supporting essentially object-oriented languages;
- it does not support tail calls – a function-calling mechanism which does not keep stack frames and is commonly used for the efficient implementation of functional languages;
- it does not support any kind of function pointers or other low-level features not essential but useful for the efficient compilation of functional languages.

In contrast, the .NET Platform was designed for supporting several programming languages paradigms, supporting a larger set of primitive types, enumerations, pointers, multidimensional arrays, arguments passing by reference, etc.. Besides this, it supports tail calls and type-safe function pointers through the use of *delegates*. Finally, .NET offers many features not found in JAVA Platform such as versioning mechanisms and seamless interoperability with other languages. Therefore it appears to be a more promising platform for the implementation of functional languages than the JVM.

3 Design Space

In this section we describe the available design space to implement elements commonly found in functional languages in object-oriented environments such as the .NET Platform.

3.1 Code Generation

There are a few possibilities to be considered for the language in which the final code will be generated for .NET. The simplest possibility is to target a language which is already supported in .NET, such as C# [C#, 2004]. The compiler of this language is then responsible for generating CIL code. This approach is used by Mondrian [Meijer et al., 2001]. Despite its simplicity, generating high-level code makes it impossible the use of some low level instructions which exist only in CIL and help to improve performance (e.g.: instruction to generate tail calls).

F# [F#, 2005] adopts a distinct approach. Its compiler generates code in ILX [Syme, 2001], an extension to CIL intended to support functional features.

We considered at first the generation of ILX code, but it would make our project dependent on the design decisions adopted by ILX's implementors, leaving little space for optimizations and further investigations on different implementations. Thus generating CIL code directly is a more flexible and efficient solution, and we have chosen this approach.

3.2 Closures Representation

Closures are dynamically allocated objects which point to a code and to a set of *free variables,* or *environment,* which is accessed by this code. An example is shown below:

```
f :: Int -> u -> (Int -> Int)
f x y = let g w = x + w in g
```

In the example, f is a function which receives 2 arguments (x and y, of type Int and u, respectively) and returns an element of type (Int -> Int). g is a closure which encapsulates a code which waits for 1 argument (w) and accesses an argument received by f (x). Here, x is a *free variable* in g, because it is not declared on g's scope. f may also be considered a closure, without free variables.

In the non-strict evaluation mechanism, also called *lazy evaluation*, the arguments received by a function are not evaluated before the call, as in most imperative and object-oriented languages, which are strict. They will be evaluated in the function's body only when (and if) their values are needed. So, each argument should be a closure pointing to the code responsible for evaluating it when necessary. In the given example, x and y are closures and f and z might be passed as parameters to any other functions.

There are several techniques to represent closures in object-oriented platforms. Here we will introduce some of these techniques which can be used in the .NET Platform.

3.2.1 One-class-per-closure

A commonly used strategy to represent closures in object-oriented platforms is the generation of one class per closure. This strategy assumes the existence of an interface or abstract class with one or more methods responsible for applying the closure's code to its arguments. Each closure is compiled to a subclass of that abstract class, the closure's compiled code is inserted into the suitable application method and the free variables are compiled to fields. An example of this technique is shown below. For a function defined as:

```
foo :: ...
foo <arguments> = /*foo's code here*/
```

and a pre-defined abstract class defined as:

```
abstract class Closure {
  ...
  public abstract object Invoke(...);
}
```

the following class is generated:

```
class foo: Closure {
   ...
   public object Invoke(...) {
      /*foo's compiled code here*/
}
```

This technique may be implemented in several ways: Invoke may receive an array of objects that represent the function's arguments or may receive no arguments at all and obtain the function's arguments from another place (eg.: a stack). Another approach is to use several overloads of `Invoke`, each one receiving a different number of arguments.

The disadvantage of this approach is the large number of classes which would be created for an average-sized program written in a non-strict language. On the .NET platform, each class has metadata which should be loaded and kept in memory. Besides this, CLR runs verification routines which may increase the general execution overhead.

This strategy is used by Mondrian (non-strict), F# (strict) and by the standard ILX implementation. F# avoids the generation of many closures by means of inline techniques. Moreover, it is a strict language and in these languages closures are less frequent.

Some implementations of functional languages for the Java Virtual Machine [Choi et al., 2001], [Bothner, 1998], [Tullsen, 1996] also used this technique.

3.2.2 Pre-defined classes

In this approach, a set of pre-defined classes is added to the compiler's runtime system and each closure is represented as an instance of one of those classes. There may be classes for closures which receive 0, 1, 2,...,n arguments. The important point here is that a large set of objects share the same class and their code could be compiled to static methods, for example, rather than to classes. The extra costs of this approach come from additional indirections for accessing the static methods and from the access to free variables, which may be stored in arrays. However the indirections should not carry out very large costs in the presence of tail calls.

In the example shown in 3.2.1, these classes might be subclasses of `Closure`.

There are several variations of this technique. They differ in the way by which the closure's code is accessed. Here we considered three alternatives:

Nonverifiable code:

.NET allows the storage of a function pointer in the closure body. In CIL there are instructions which allow the direct manipulation of function pointers, but at a cost: the resulting code is not verifiable, which means it is not guaranteed to be type-safe. According to [Box and Sells, 2003], "the ability to load nonverifiable code is itself a permission that one must explicitly grant to code through policy. The default policy that is installed with the CLR grants this permission only to code that is installed on a local file system". So downloaded code that is not verified cannot be easily executed. Obviously this restriction is a disadvantage of this strategy, and future versions of .NET Platform intend to impose even higher restrictions to nonverifiable code. However this appears to offer good performance and can be offered to the user as an optional feature.

The use of nonverifiable code was originally adopted by ILX but it was soon abandoned. At the current implementation, ILX uses the one-class-per-closure technique described above.

Delegates:

Delegates, a new feature introduced with the .NET Platform, are the type-safe version of function pointers. They are objects which encapsulate a pointer to a method to be invoked and its target object. By using this .NET feature, each closure could be a delegate.

We have not found any implementation using delegates for the representation of closures due to the performance problems found in their implementation on the first versions of .NET. However we executed some experiments that showed that delegates' performance has been improved significantly on .NET 2.0. Therefore we decided to use them for closures representation in Haskell .NET.

"Indexed code":

This approach is adopted by Bigloo for Scheme, a strict language, and is described in [Bres et al., 2004]. It is also used by some implementations that target the JVM, such as [Serpette and Serrano, 2002] and [Wakeling, 1999]. The use of the indexed code technique consists of storing an integer index in each closure and, when it is invoked, a test on the value stored in the index is carried out and the suitable function is invoked. Each function is mapped to a different index and the closure pointing to the function stores the value of this index.

Performance tests carried out by Bigloo's implementors showed superior performance through this strategy rather than through the use of delegates. However the tests were based on the first versions of .NET Framework. Nonetheless, we intend to compare the performance of this technique with the performance achieved by delegates in .NET 2.0 in the future.

3.3 Applications

The implementation of function applications in functional languages is not so trivial as in imperative and object-oriented paradigm. In functional languages functions are first-class values that can be passed as parameters, returned from other functions and stored in data structures. It means that a statically unknown function may be applied to any number of arguments, even to a number of arguments different from its original arity.

There are two execution models which deal with function application: push/enter and eval/apply. In the eval/apply execution model, the code which calls the function is responsible for inspecting its arity and applying the function to the correct number of arguments. If a function f is called with less arguments than necessary, a *partial application* is constructed and returned. On the other hand, if it is called with more arguments than necessary, we call it assuming that the function returned by f will consume the remaining arguments. The standard function call mechanism – that uses the CLR's stack for passing parameters – can be used even for functions which are not known statically. This model is used in F#, ILX and Bigloo for Scheme.

In the push/enter execution model, the called function's code is responsible for identifying its arity and for applying it to the correct number of arguments. When an unknown function or a function applied to an insufficient number of arguments is called, the arguments are pushed onto a stack and its code is "entered". However, if the function is known statically and is applied to enough arguments, the standard function call mechanism can be used. The push/enter model is used by Mondrian and [Choi et al., 2001].

Simon Marlow and Simon Peyton Jones [Marlow and Jones, 2004] gave a formal definition for the two mechanisms and compared their implementations when generating C code and using the Glasgow Haskell Compiler [Jones et al., 93]. The final conclusion is that both models present similar performance in that context.

We cannot say which one is better in the .NET. To answer this question, both models need to be implemented and compared. In eval/apply unnecessary partial applications may be instantiated and extra tests on the function arity need to be performed. In push/enter, a stack must be built and managed.

We have adopted push/enter for the first version of Haskell .NET. We intend to implement eval/apply in future versions in order to evaluate which execution model performs better in the .NET Platform.

3.4 Parametric Polymorphism

A common feature in functional languages is the ability to define functions which can receive arguments of any type.

In strongly typed platforms such as .NET and JVM, a traditional approach to solve this problem is the use of a common superclass to represent the polymorphic argument's type. In .NET, this type can be `System.Object`, or even a more specific type defined by the compiler's runtime system (e.g: `Closure`). The use of a common superclass is adopted by Mondrian.

Another possible approach is the use of *Generics* [Kennedy and Syme, 2001], a new feature incorporated into .NET 2.0 and JAVA 1.5. It aims to correct some limitations of the above technique by avoiding some runtime errors and unnecessary casts. Generics is currently used by F#.

Since our implementation uses non-strict evaluation, the polymorphic arguments are always typed as closures. Therefore the use of Generics would not give us any significant performance benefits.

3.5 Discriminated Unions

In functional languages we often define data types such as:

$$\mathsf{data\ List\ t} = \mathsf{Nil} \mid \mathsf{Cons\ t\ (List\ t)}$$

We call these definitions *discriminated unions*. In the above example, Nil and Cons are *constructors*.

In .NET there is not a direct counterpart for discriminated unions. Instead we have enumerations, but these types cannot receive arguments and are mapped to integer constants. However, we have the concept of classes, which can be used to represent record types and subtyping relationships. We only need to find efficient ways to inspect which constructor is being used.

One intuitive option is the generation of an abstract class to represent the discriminated union – in the above example, a class called `List` – and one subclass per constructor. We can then use a special CIL instruction to dynamically discover the object's class. An example is shown below, in C# syntax:

```
List l;
...
if(l is Nil){...}
else if(l "is Cons){...}
```

In the example, the operator **is** is compiled to the special CIL instruction cited in the last paragraph. However, an even simpler approach is to annotate each object with an integer tag that identifies its constructor [Jones, 1987]. This way we can dynamically inspect this tag without using any special instructions, like in the example shown below:

```
switch(l.tag){
  case 1: /* 1 is the tag corresponding to
          Nil in the type definition*/
          ...
  case 2: /* 2 is the tag corresponding to
          Cons in the type definition*/
          ...
}
```

The test for an integer constant tends to be cheaper than the test for an object type. Besides this, a switch instruction is usually implemented by a lookup table instead of a set of sequential tests. In general, this technique showed better performance in our experiments and we decided to use it for the Haskell .NET compiler. Finally, we avoid the generation of classes by having a set of pre-defined classes for each number of constructor arguments.

4 The Haskell .NET compiler

In this section we introduce the Haskell .NET compiler, a Haskell implementation targeting the .NET Platform.

We do not aim to achieve excellent execution times but only a sufficiently good performance to enable future experimentation with interoperability in .NET, one of our long-term objectives. Such interoperability will make it possible to invoke routines written in other .NET languages from Haskell and vice-versa.

We used an optimizing state-of-art Haskell compiler – the Glasgow Haskell Compiler (GHC) – as a basis for our work. Particularly, we have added to GHC (version 6-2.2) one back-end capable of generating CIL code.

In the next subsections we introduce the proposed compilation strategies.

4.1 Basic Compilation Strategy

We decided to adopt a compilation strategy mostly based on the *Spineless Tagless G-Machine* (STG) [Jones, 1992], a compilation model considered the state-of-art for implementation of non-strict functional languages.

According to the STG model, the source program is translated into a program expressed in a functional intermediate language – the *STG Language* – whose (simplified) grammar is shown in Figure 1. The output program is then compiled according to the STG machine's compilation rules.

The decision of using STG instead of GHC Core language as the source language for our compiler was motivated by the fact that the first one identifies explicitly the closures, so providing useful information to our implementation. The syntax of STG language used by GHC is not identical to the syntax shown in Figure 1 because it contains annotations related to types, strictness analysis and several other kinds of information. Particularly, the type annotations enable us to give exact types to data, reducing the number of casts, for example. All the information provided by the annotations may be used in the future for optimizations. We also consider to implement a pre-processor to remove annotation code related to specific GHC versions.

Program	$prog \rightarrow$	$binds$	
Bindings	$binds \rightarrow$	$var_1 = lf_1; ...; var_n = lf_n$	$n \geq 1$
Lambda-forms	$lf \rightarrow$	$vars_f \backslash \pi\ vars_a \Rightarrow expr$	
Update flag	$\pi \rightarrow$	u	Updateable
	$\mid$	n	Not updateable
Expression	$expr \rightarrow$	**let** *binds* **in** *expr*	Local definition
	$\mid$	**case** *expr* **of** *alts*	Case recursion
	$\mid$	*var atoms*	Application
	$\mid$	*constr atoms*	Saturated constructor
	$\mid$	*prim atoms*	Saturated built-in op
	$\mid$	*literal*	
Alternatives	$alts \rightarrow$	$aalt_1; ...; aalt_n; default$	$n \geq 1$ (Algebraic)
	$\mid$	$palt_1; ...; palt_n; default$	$n \geq 1$ (Primitive)
Algebraic alt	$aalt \rightarrow$	$constr\ vars \Rightarrow expr$	
Primitive alt	$palt \rightarrow$	$literal \Rightarrow expr$	
Default alt	$default \rightarrow$	$var \Rightarrow expr$	
	$\mid$	**default** $\Rightarrow expr$	
Literals	$literal \rightarrow$	0# |1# |...	Primitive integers
	$\mid$	...	
Primitive ops	$prim \rightarrow$	+# |-# |*# |/#	Primitive integer ops
	$\mid$	...	
Variable lists	$vars \rightarrow$	$\{var_1, ...var_n\}$	$n \geq 0$
Atom lists	$atoms \rightarrow$	$\{atom_1, ...atom_n\}$	$n \geq 0$
	$atom \rightarrow$	$var\ \mid literal$	

Figure 1: STG Language

First of all, each Haskell module is compiled to a .NET *assembly* - a kind of compilation unit. This assembly contains the declaration of a public class which has the same name as the module. The class declares public static methods representing the functions and public static fields representing the top-level closures. An assembly can access all the public classes/fields/methods declared in other assemblies. So separate compilation comes for free.

In our compilation strategy, we avoided the generation of one class per closure and adopted the use of delegates for representing closures and functions. In fact, each function/closure is compiled to two static methods: a *slow entry point* and a *fast entry point*.

The fast entry point should be invoked when we know the function that will be called and have all its arguments at hand. It uses the CLR's stack to find its arguments, enabling thus the use of efficient argument passing mechanisms. The slow entry point is used in two situations: when the function is applied to an insufficient number of arguments (partial application) or when we do not know the arity of the function to be called, which occurs when it is an argument or is stored in a data structure, for example.

The slow entry point receives only one argument: the closure, which contains free variables if they exist. In these cases, the function's arguments are expected to be in special stacks referred here as"HSNET's stacks" [1]. We maintain array based data structures to implement this kind of stack and keep one stack per possible type – integer, floating point number, closure or generic object. The slow entry point checks the sizes of the HSNET's stacks of the corresponding arguments types. If there are enough arguments on it, they are popped and pushed on the CLR's stack and the fast entry point is called through a tail call instruction. Otherwise, the arguments found on the HSNET's stacks are popped and stored into the the closure argument, which is then returned. When a known function is called with more arguments than necessary, only the excess arguments are pushed on the HSNET's stacks and the fast entry point is called.

An example of this strategy is shown in the next section. Appendix A shows the compilation schemes used in the current implementation.

4.2 Runtime System

In order to support the representation of closures, we needed to provide a runtime system with the necessary types and the HSNET's stacks access methods. In Figure 2 we show a subset of the types provided by our runtime system. The HSNET's stack access methods are encapsulated in a separated module and are not showed here for simplicity.

In our runtime system, the closure's Enter method is responsible for the invocation of the closure's slow entry-point, passing to it the reference to the closure object to enable the closure's code to access the free variables.

It is time we examined in details .NET delegates and their Invoke method. A delegate inherits from .NET System.Delegate class and stores a pointer to a method as well as a pointer to the object in which the method will be called (or a null pointer in the case of static methods). Whenever a delegate's Invoke method is called, the method pointed by this delegate is called. Invoke's signature must match exactly the signature of the method pointed by the delegate. Invoke's

[1] Note that this is not the CLR's stack

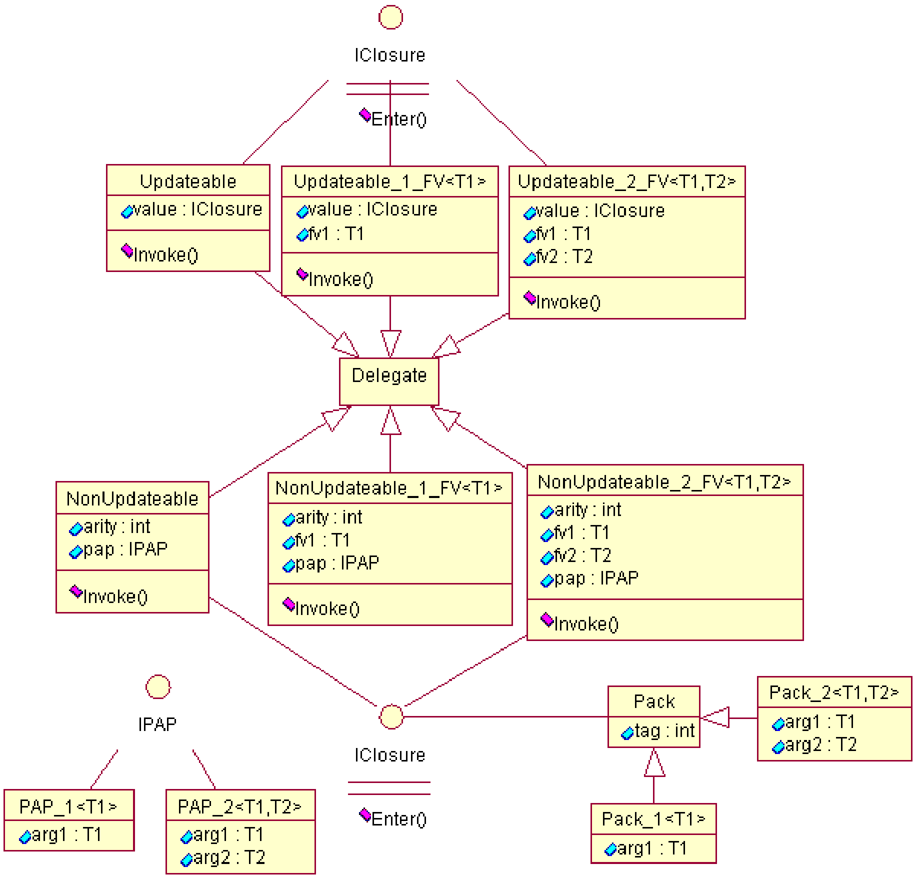

Figure 2: Runtime System

body is implemented by the CLR and consists basically of a jump to the pointed method's body [Box and Sells, 2003]. In our implementation, a delegate points to the closure's slow entry point, and Invoke is called by the Enter method. Although it is not shown in Figure 2 due to lack of space, the Invoke method receives as argument the closure (referenced in the Enter method through the "this" pointer).

As it can be seen from Figure 2, two kinds of closures are treated differently: updateable closures and non-updateable closures. Updateable closures point to functions with no arguments and are updated in order to avoid the same computation more than once. Non updateable closures usually point to functions which receive arguments. The analysis responsible for identifying updateable closures is done statically by the STG code generator.

The updateable closures keep a field – value – that stores the closure's value after its first evaluation and its Enter method's implementation follows the steps below:

1. Tests if the closure's value was already computed by inspecting the value field. If it was already evaluated, a tail call to value's Enter method is done. Otherwise, step 2 is executed.

2. Pushes the HSNET's stacks' sizes on a second stack – the *update stack* – and sets their sizes to 0. This step has the same effect as saving the update frame in the STG machine.

3. Generates a call to Invoke and stores the result in value.

4. Restores the HSNET's stacks' sizes.

The non-updateable closures' Enter method just makes a tail call to the Invoke method. They also keep an object which encapsulates arguments previously passed in partial applications – the "pap" field. It stores the received arguments for future applications of the same closure/function. Finally, the arity field is decremented by the same number of elements which are stored in the partial application field.

In the current implementation, we obtain the arity information from the STG program generated by GHC. When a function/closure is statically unknown, its arity information is also unknown and is set to 0 (zero) by GHC's front end. This leads us to deal with calls to unknown closures and to updateable closures (arity = 0) in the same way: generating a call to the slow entry point (the closure's Enter method).

Again from Figure 2 we see classes that represent data instead of closures – the "Pack..." classes [Jones, 1987]. Instances of these classes represent constructors of discriminated unions. They store the constructor's arguments and a tag to identify it. Their Enter method simply returns. Some optimizations are done, such as mapping nullary constructors applications to null whenever possible.

In order to represent the types of elements such as free variables, function arguments and constructor arguments, we use Generics. The Figure 2 shows only a subset of our runtime system, because in fact there are classes for representing closures with more than 2 arguments or more than 2 free variables and data constructors with more than 2 arguments. For situations in which a very large number of arguments/free variables are needed, new classes can be generated.

The compilation strategy is better understood with an example: the map function, which receives a function f and a list l and returns a new list whose elements result from the application of f to the elements of l. The STG code for map is shown below. We use the STG notation $vars_f \backslash \pi \ vars_a$ -> $expr$ ($vars_f$ is the set of free variables, $vars_a$ is the set of arguments and π is the update flag – u for updateable and n for non-updateable).

```
map = {} \n {f,l}-> case l of
          Nil {}-> Nil {}
          Cons {x,xs }-> let fx = {f,x} \u {} -> f {x}
                                fxs = {f,xs} \u -> map {f,xs}
                             in Cons {fx,fxs}
```

The code generated is similar to the one shown below, written in pseudo-C# notation for simplicity. In the code, "`<tail>`" means tail call.

```
//slow entry point
public static IClosure map(NonUpdateable clo){
    /*If there are enough arguments in the HSNET's stacks, tail calls the fast
    entry point. Otherwise, returns a partial application.*/
}
//fast entry point
public static Pack_2<IClosure, IClosure >map(IClosure f, IClosure l){
  Pack_2<IClosure, IClosure> scrutinee = (Pack_2<IClosure, IClosure>)l.Enter();
  if(scrutinee != null) {
    IClosure x = scrutinee.args1;
    IClosure xs = scrutinee.args2;
    Updateable_2_FV<IClosure, IClosure> fx_closure =
        new Updateable_2_FV<IClosure, IClosure>(fx);
    fx_closure.fv1 = f;
    fx_closure.fv2 = x;
    Updateable_2_FV<IClosure, IClosure> fxs_closure =
        new Updateable_2_FV<IClosure, IClosure>(fx);
    fxs_closure.fv1 = f;
    fxs_closure.fv2 = xs;
    return new Pack_2<IClosure, IClosure>(fx_closure, fxs_closure);
  } else return null;
}
public static IClosure fx(Updateable_2_FV<IClosure, IClosure> closure){
  RTS.Push(closure.fv2);
  <tail>closure.fv1.Enter(); //call to unknown function
}
public static IClosure fxs(Updateable_2_FV<IClosure, IClosure>closure)
  <tail>map(closure.fv1, closure.fv2);
}
```

fx and fxs do not need fast entry-points since they have no arguments.

5 Conclusions and Future Work

We have described a compilation strategy for non-strict functional languages targeting the .NET Platform. Our strategy was based on the push/enter execution model and we focused on the Haskell language as a case study. Our approach does not generate one class per closure/function like most non-strict functional languages' implementations for object-oriented virtual machines. Instead, it takes advantage of a new feature present in .NET: delegates, a kind of type-safe function pointer. Like closures, delegates are dynamically allocated objects that can store fields and a pointer to code. Finally, we support efficient argument passing mechanisms and tail calls.

Our implementation is in an initial stage, currently supporting the compilation of part of the Haskell prelude and small programs. We have not evaluated and tuned it for performance yet, and therefore it is too early to present performance results compared to other Haskell compilers or GHC back-ends.

A question to be addressed is the impact of runtime typing, because it may generate considerable overhead due to unnecessary type verifications. Functional languages are statically typed and so runtime type errors should never occur. We have not measured this impact yet. In performance critical applications which do not have security restrictions, it can be reduced through the generation of unverifiable code, where runtime casts are not mandatory. However, we believe that some extensions should be incorporated into the CLR in order to support static typed languages in an efficient and type-safe way.

We plan to experiment with the eval/apply execution model. It enables us to remove the HSNET's stacks. However, it may lead to the instantiation of more objects representing partial applications and extra tests on function arity. We will be able to decide which execution model is best for the .NET Platform after implementing both and comparing their performance over different aspects. We also intend to experiment with other representations for closures in order to find out if delegates really lead to the best performance. These experiments may tell us if extensions to the CLR architecture are necessary. We believe that the native CLR support to the representation of closures might provide better performance, but only experimentation can give us an exact answer.

We will also work on interoperability issues for Haskell .NET by researching ways of calling Haskell code from other .NET languages and vice-versa. The interoperability with other .NET languages and consequently with .NET standard libraries will enable the Haskell programmers to implement real-world systems, including graphical user interfaces, distributed systems and web applications. At the present, the interoperability with the .NET world occurs at a low level, in the implementation of standard I/O functions from the Haskell prelude. Finally, we intend to develop the necessary tools for enabling the development of ASP .NET pages in Haskell.

6 Related Work

Some well known strict functional languages have been successfully implemented in the .NET Platform, such as SML [Benton et al., 2004] – extended to provide interoperability with .NET – and Scheme (Bigloo) [Bres et al., 2004]. The first one was extended to provide interoperability with .NET. and provides a limited support to separate compilation.

Mondrian is a non-strict functional language which was implemented for the .NET Platform. However its implementation was highly experimental and had no optimizations. Since it targets C#, it does not support tail-calls, for example.

Other attempts have been made to compile Haskell to .NET. One of them consisted of compiling Haskell to Mondrian code, but it did not cover the full Haskell prelude. Another attempt consisted of generating ILX code. However, both implementations had no efforts towards performance and were mostly proofs of concept. None of them have published performance results.

Finally, some solutions resulted in the creation of new languages, such as Mondrian and F#. However, the problem of creating new functional languages targeting the .NET Platform is that they are not known by the development community. Besides this, they tend to incorporate hybrid or multi-paradigm features and sacrifice the typical functional features, in some cases. Nonetheless some of these languages have very interesting interoperability characteristics, being free to define their syntax and semantics.

Acknowledgements

We would like to thank Microsoft Research, FACEPE and CNPq Brazilian Research Agency for the project funding. Special thanks to all the people who helped us with useful information about related projects and GHC, especially Don Syme, Simon Peyton Jones and Sigbjorn Finne. Finally, we should thank the SBLP 2005 anonymous referees who contributed with suggestions for the improvement of this paper.

References

[Benton et al., 2004] Benton, N., Kennedy, A., and Russo, C. V. (2004). Adventures in Interoperability: The SML.NET Experience. In *6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 215–226, Verona, Italy. ACM Press.

[Bothner, 1998] Bothner, P. (1998). Kawa: Compiling Scheme to Java. In *Lisp Users Conference*, Berkeley, California, USA.

[Box and Sells, 2003] Box, D. and Sells, C. (2003). *Essential .NET: Volume 1 - The Common Language Runtime*. Addison-Wesley Pearson Education, Boston.

[Bres et al., 2004] Bres, Y., Serpette, B. P., and Serrano, M. (2004). Compiling Scheme programs to .NET Common Intermediate Language. In *.NET Technologies'2004 Workshop*, Plzen. ©UNION Agency-Science Press.

[C#, 2004] C# (2004). Hyperlinked ECMA C# Language Specification. Available at: http://www.jaggersoft.com/csharp_standard/. Last visited on 14 fev. 2005.

[Choi et al., 2001] Choi, K., il Lim, H., and Han, T. (2001). Compiling Lazy Functional Programs Based on the Spineless Tagless G-Machine for the Java Virtual Machine. In *Functional and Logic Programming: 5th International Symposium, FLOPS 2001*, volume 2024 of *Lecture Notes in Computer Sciences*, page 78, Tokyo, Japan. Springer-Verlag GmbH.

[ECMA, 2002] ECMA (2002). *Final Draft Common Language Infrastructure (CLI) - Partition III: CIL Instruction Set*. ECMA. Final draft produced by ECMA TC39/TG3. Available at http://download.microsoft.com/download/6/8/8/68863d89-d35d-4bc5-8a1c-7e0a02e1881e/Partition_III_CIL.zip.

[F#, 2005] F# (2005). F#. Available at: http://research.microsoft.com/projects/ilx/fsharp.aspx. Last visited on 09 fev. 2005.

[Gosling et al., 2000] Gosling, J., Joy, B., Steele, G. L., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley Longman, 2nd edition.

[Haskell, 2002] Haskell (2002). The Haskell 98 Language Report. Available at: http://www.haskell.org/onlinereport/. Last visited on 03 mar. 2005.

[Jones, 1987] Jones, S. L. P. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall International series in computer science. Prentice Hall, Hertfordshire, UK.

[Jones, 1992] Jones, S. L. P. (1992). Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. *Journal of Functional Programming*, 2(2):127–202.

[Jones et al., 93] Jones, S. L. P., Hall, C. V., Hammond, K., Partain, W., and Wadler, P. (93). The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*.

[Kennedy and Syme, 2001] Kennedy, A. and Syme, D. (2001). Design and Implementation of Generics for .NET Common Language Runtime. Technical report, Microsoft Research, Cambridge, UK.

[Lindholm and Frank, 1999] Lindholm, T. and Frank, Y. (1999). *The Java Virtual Machine Specification*. Addison-Wesley Longman, 2nd edition.

[Marlow and Jones, 2004] Marlow, S. and Jones, S. L. P. (2004). Making a fast curry: Push/enter vs eval/apply for higher-order languages. In *Proc. International Conference on Functional Programming*, Snowbird.

[Meijer et al., 2001] Meijer, E., Perry, N., and van Yzendoorn, A. (2001). Scripting .NET Using Mondrian. *Lecture Notes in Computer Science*, 2072:150–164.

[Serpette and Serrano, 2002] Serpette, B. and Serrano, M. (2002). Compiling Scheme to JVM Bytecodes: A Performance Study. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, pages 259–270, New York. ACM, ACM Press.

[Syme, 2001] Syme, D. (2001). ILX: Extending the .NET Common IL for Functional Language Interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1).

[Tullsen, 1996] Tullsen, M. (1996). Compiling Haskell to Java. Technical report, Department of Computer Science – Yale University, New Heaven.

[Wakeling, 1999] Wakeling, D. (1999). Compiling lazy functional programs for the Java Virtual Machine. *Journal of Functional Programming*, 9(6):578–603.

A Compilation Schemes

The set of compilation rules used by the Haskell .NET compiler is presented in Figures 3 and 4. The expression currently being compiled is shown in the left hand-side, in STG notation, and its compiled version is shown in the right hand-side. Although the compiled version is in fact generated in CIL, it is shown in C# for simplicity.

The following schemes are used in the compilation rules:

- **B** (Figure 3), responsible for the compilation of bounded expressions,
- **E** (Figure 3), responsible for the compilation of expressions,
- **A** (Figure 4), responsible for the compilation of alternatives,
- **C** (Figure 4), responsible for the compilation of auxiliary expressions,
- **T** (Figure 4), responsible for returning the corresponding .NET type of an STG type,
- **L** (Figure 4), responsible for returning the lazy type corresponding to an STG type. The "lazy type" is meant to be *closure type* (interface IClosure) if the STG type is non-primitive, or the corresponding .NET primitive type otherwise.

Other symbols are also used in the compilation schemes:

- τ is the STG type of an element;
- * is used for an optional element;
- "SEP" is an abbreviation for "slow entry-point" and <<id>_SEP> means "the slow entry-point of the function bounded to id". In C# language it is the method's name, however in CIL a pointer to the method is used instead.
- "push" is an abbreviation for the method responsible for pushing an object on the suitable HSNET's stack.

Although it is not explicitly shown in the compilation schemes, a function invocation is compiled to a tail call whenever possible. However, if a function invocation is not the last expression in its scope, the tail call obviously cannot be used and a normal call is generated instead.

For simplicity, we do not include in the compilation schemes the code relative to the slow entry points nor the declarations of the generated methods. Moreover, this code is not directly related to the expressions shown in the first column.

$\mathbf{B}[prog \rightarrow bind_1, ..., bind_n]$	$= \mathbf{B}[bind_1]...\mathbf{B}[bind_n]$
$\mathbf{B}[id = f_1..f_m \backslash n\ a_1..a_n \rightarrow\ expr]$	= NonUpdatable_FV_<m> <T[$\tau(f_1)$],...,T[$\tau(f_m)$]> id = new NonUpdatable_FV_<m> <T[$\tau(f_1)$],...,T[$\tau(f_m)$]>(<id_SEP>); id.fv<1>=$\mathbf{C}[f_1]$; ... id.fv<m>=$\mathbf{C}[f_m]$; id.arity = <n>;
$\mathbf{B}[id = f_1..f_m \backslash u\ \rightarrow\ expr]$	= Updateable_FV_<m> <T[$\tau(f_1)$],...,T[$\tau(f_m)$]> id = new Updateable_FV_<m> <T[$\tau(f_1)$],...,T[$\tau(f_m)$]>(<id_SEP>); id.fv<1>=$\mathbf{C}[f_1]$; ... id.fv<m>=$\mathbf{C}[f_m]$;

$\mathbf{E}$[let *bind* in *expr*]	= $\mathbf{B}[bind]$; $\mathbf{E}[expr]$
$\mathbf{E}$[let $bind_1, ..., bind_n$ in *expr*]	= $\mathbf{B}[bind_1]$;...;$\mathbf{B}[bind_n]$; $\mathbf{E}[expr]$
$\mathbf{E}$[case *expr* of *alts*]	= $\mathbf{T}[\tau(expr)]$ scrutinee = ($\mathbf{T}[\tau(expr)]$) $\mathbf{E}[expr]$; $\mathbf{A}[alts]$
$\mathbf{E}[var\ \ atom_1...atom_n, atom_{n+1}, ..., atom_m]$, *var* has **no** free variables and has arity n, $m \geq 0$	= push $\mathbf{C}[atom_m]$; ...; push $\mathbf{C}[atom_{n+1}]$; var ($\mathbf{C}[atom_1], ..., \mathbf{C}[atom_n]$);
$\mathbf{E}[var\ \ atom_1...atom_n, atom_{n+1}, ..., atom_m]$, *var* has free variables and has arity n, $m \geq 0$	= push $\mathbf{C}[atom_m]$; ...; push $\mathbf{C}[atom_{n+1}]$; var (var, $\mathbf{C}[atom_1], ..., \mathbf{C}[atom_n]$);
$\mathbf{E}[var\ \ atom_1...atom_m]$, *var* has arity n, $m < n$	= push $\mathbf{C}[atom_m]$; ...; push $\mathbf{C}[atom_1]$; var.Enter();
$\mathbf{E}[var\ \ atom_1...atom_m]$, *var* has unknown arity or has arity 0, $m \geq 0$	= push $\mathbf{C}[atom_m]$; ...; push $\mathbf{C}[atom_1]$; var.Enter();
$\mathbf{E}[constr_i]$, if $constr_i$ is the only nullary constructor	= null
$\mathbf{E}[constr_i\ \ atom_1...atom_n]$	= new Pack_<n> <$\mathbf{L}[\tau(atom_1)]$,...,$\mathbf{L}[\tau(atom_n)]$> (i, $\mathbf{C}[atom_1], ..., \mathbf{C}[atom_n]$)
$\mathbf{E}[\Theta\ atom_1...atom_n]$, Θ is primitive	= $\mathbf{C}[atom_1]\ \Theta\ ...\ \Theta\ \mathbf{C}[atom_n]$
$\mathbf{E}[n\#]$, where n# is a literal of a primitive type	= n

Figure 3: $\mathbf{B}$ and $\mathbf{E}$ Compilation Schemes

$\mathbf{A}[constr_1\ arg_0...arg_m \rightarrow expr_1;$
...;
$constr_i \rightarrow expr_i;$
...;
$constr_n\ arg_0...arg_p \rightarrow expr_n;$
$default]$
if $constr\_i$ is the only nullary constructor
	= if(scrutinee == null){
	$\mathbf{E}[expr_i]$
	} else {
	switch(scrutinee.tag){
	case $constr_1$'s tag:
	arg_1 = scrutinee.arg1;
	...
	arg_m = scrutinee.arg<m>;
	$\mathbf{E}[expr_1]$...
	case $constr_n$'s tag:
	arg_0 = scrutinee.arg1;
	...
	arg_p = scrutinee.arg<p>;
	$\mathbf{E}[expr_n]$
	$\mathbf{A}[default]$ }

$\mathbf{A}[constr_1\ arg_0...arg_m \rightarrow expr_1;$
...;
$constr_n\ arg_0...arg_p \rightarrow expr_n;$
$default]$
	= switch(scrutinee.tag){
	case $constr_1$'s tag:
	arg_1 = scrutinee.arg1;
	...
	arg_m = scrutinee.arg<m>;
	$\mathbf{E}[expr_1]$...
	case $constr_n$'s tag:
	arg_0 = scrutinee.arg1;
	...
	arg_j = scrutinee.arg<p>;
	$\mathbf{E}[expr_n]$
	$\mathbf{A}[default]$
	}

$\mathbf{A}[\text{default} \rightarrow expr]$	= case default: $\mathbf{E}[expr]$;
$\mathbf{A}[var \rightarrow expr]$	= case default: $\mathbf{C}[var]$ = scrutinee; $\mathbf{E}[expr]$;
$\mathbf{A}[literal_1 \rightarrow expr_1;$ ...; $literal_n \rightarrow expr_n;$ $default]$	= switch(scrutinee){ case $\mathbf{E}[literal_1]$: $\mathbf{E}[expr_1]$... case $\mathbf{E}[literal_n]$: $\mathbf{E}[expr_n]$ $\mathbf{A}[default]$ }

$\mathbf{C}[var]$	= var
$\mathbf{C}[literal]$	= $\mathbf{E}[literal]$
$\mathbf{L}[int\#] = \mathbf{T}[int\#]$	= `int32`
$\mathbf{L}[char\#] = \mathbf{T}[char\#]$	= `char`
$\mathbf{L}[float\#] = \mathbf{T}[float\#]$	= `float32`
$\mathbf{L}[t]$, t is a boxed type	= `IClosure`
$\mathbf{T}[t]$, where t has the form data X = ... $\mid\ constr_i\ t_1...t_m$ $\mid$ ... $\mid\ constr_j\ u_1...u_n$ $\{t_1, ...t_n\} \neq \{u_1, ...u_n\}$	= `Pack`
$\mathbf{T}[t]$, where t has the form data $X = (constr_i)*$ $\mid\ constr_j\ \ u_1...u_n$ $\mid$ ... $\mid\ constr_k\ \ u_1...u_n$	= `Pack_`<n><$\mathbf{L}[u_1]$,...$\mathbf{L}[u_n]$>

Figure 4: A, C, L and T Compilation Schemes