ILX: Extending the .NET Common IL for Functional Language Interoperability

Don Syme

Microsoft Research,
Cambridge,
dsyme@microsoft.com

Abstract

This paper describes several extensions to the .NET Common Intermediary Language (CIL), each of which is designed to enable easier implementation of typed high-level programming languages on the .NET platform, and to promote closer integration and interoperability between these languages. In particular we aim for easier interoperability between components whose interfaces are expressed using function types, discriminated unions and parametric polymorphism, regardless of the languages in which these components are implemented. We show that it is possible to add these constructs to an existing, "real world" intermediary language and that this allows corresponding subsets of constructs to be compiled uniformly, which in turn will allow programmers to use these constructs seamlessly between different languages. In this paper we discuss the motivations for our extensions, which are together called Extended IL (ILX), and describe them via examples. In this setting, many of the traditional responsibilities of the backend of a compiler must be moved to ILX and the execution environment, in particular those related to representation choices and low-level optimizations. We have modified a Haskell compiler to generate this language, and have implemented an assembler that translates the extensions to regular or polymorphic CIL code.

1 Introduction

Given a world with many programming languages, we need the ability to translate constructs between languages, or, alternatively, we need common subsets of language constructs that make sense in most or all languages. Recently there has been more emphasis on langugage implementations that follow the second path [27, 1, 5, 8]. This paper describes the design and implementation of one such common set of constructs, and its realization as a set of concrete extensions to the input language of an execution platform that has already been designed with some of this kind of interoperation in mind. The platform is the .NET Common Language Runtime [5], an implementation of which is provided by Microsoft as part of VisualStudio.NET. The intermediary language of the CLR is called Common IL (CIL), and our extended language is called ILX.

The constructs that we have included in ILX so far and which we describe in this paper are:

- First-class functions, closures and thunks;
- Parametric polymorphism;
- Discriminated unions;
- First-class type functions.

Our aim is to provide a practical IL that allows for a standardized treatment of constructs found in many typed high-level programming languages, e.g. Mercury, O'Caml and Generic C# [9, 19, 14]. This is the first time a widely supported IL has been modified to support these constructs. Similarly this is the first time these constructs have been integrated into a typed, object based IL of the kind supported by the JVM [20] or the .NET CLR.

We have successfully modified the GHC Haskell Compiler [15] to target these extensions.[1] The compiler produces ILX modules with predictable, typed interfaces that could easily be called from other .NET languages. Implementors of Mercury, Scheme and Vault [7, 8, 9] are actively pursuing the use of ILX to compile some or all of the corresponding constructs for their languages. We are planning an implementation of the core languages of Standard ML and OCaml.

The remainder of this section discusses the motivations and design aims for ILX. §2 describes the .NET CLR, and §3-5 cover the new constructs, describing them mainly by example. §6 discusses related work and we conclude in §7. The Appendix describes the current ILX SDK and an aspect of the current implementation required to support cross-module recursion and referencing.

We only give brief discussions on implementation techniques in each section as it we believe it is essential to focus on the *design* of ILX, and not just the performance properties of its implementation. Getting functional languages to run fast has been well explored, but no one has successfully provided a basis for getting them to talk to each other. Clearly this can only be done if a design is suitable for the needs of many languages.

1.1 Motivation

In a multi-language component programming environment such as .NET, language interoperability is one of the key determinants of programmer productivity. There are several possible ways to reduce overheads associated with language interoperability. One, which is certainly attractive when it is practical, is to orient the entire computing environment around a single programming language. This approach will naturally be the one preferred by any single programmer, who will have a favorite language that he or she

[1] Though see the caveats in the conclusion.

thinks the whole world should be using, or if a project is starting from scratch without legacy constraints. However, for better or for worse the computing world is very multilingual – this has certainly been the case historically and there are of course many languages in use to day. There are many reasons to expect this situation to continue: the existing investments in today's languages are enormous – it is expensive to change existing source code bases, or to retrain programmers, or to re-implement advanced compilation technology. This has been the primary motivation for Microsoft's approach to multi-language programming in the context of the .NET platform.

This situation poses both opportunities and problems for languages with functional and algebraic constructs (e.g. Standard ML, O'Caml, Mercury, Scheme and Haskell). The opportunity is that if language interoperability issues are solved, then these languages are better placed to be used in the areas where they are most suited, and components can be written in these languages without revealing that "exotic" techniques are being used.

However there are many problems. Foremost amongst these is the simple fact that the vast majority of libraries are now being written using APIs expressed in the Java and .NET object systems. Recently some progress has been made in this area [1, 9, 16, 27]. But the problems for these languages go beyond simply being able to access libraries: there is a fundamental lack of interoperability between these languages over the very constructs which give higher programmer productivity, i.e. the functional and algebraic constructs such as polymorphism and discriminated unions. Interoperability is hard for this feature set for the simple reason that there are a number variations on these constructs, and for each there are many ways to implement them. For example, there are many ways to represent discriminated unions in an object model, and even more ways to represent them at the level of bits and bytes provided by COM and Corba, or a C FFI. Unfortunately it is only natural that if multiple possibilities exist, then different implementations inevitably choose different possibilities, and interoperability becomes very difficult.

A related problem is that of runtime support, which is partly addressed by platforms such as the CLR. Consider garbage collection (GC): the constructs we are interested work best when the objects are stored in a GC'd heap, but GC has traditionally been a service provided by the language implementation itself. Combining multiple GCs and heaps is technically difficult and it becomes hard to trade objects seamlessly – at least some level of wrapping is typically required. Thus sharing a GC between multiple languages is a great first step. However, the services provided by the CLR are not always sufficient, as in the case of parametric polymorphism [17].

1.2 The ILX Design Aims

Given this context, ILX attempts to solve some of these problems, at least for the .NET platform. The "big-picture" aims for ILX can be summarized as follows:

- Permit practical interoperability between functional languages on the .NET platform;
- Be the easiest route to implement a new language that contains the constructs supported by ILX.

The first aim imagines a world where all the .NET languages with functional-like constructs can interoperate with relative ease. Ideally, it shouldn't matter which language a component is written in – for example Mercury, O'Caml or Standard ML components should be essentially interchangeable, as long as the APIs to the components are written in a sufficiently common fragment of the respective languages. Furthermore, for the purposes of interoperability it shouldn't matter how the common constructs (e.g. function types and closures) are represented in terms of the underlying soup of objects, code pointers and the like.

It must be admitted that both of the "big-picture" aims described above will be very difficult to realize in practice, primarily for economic and social reasons: there is already a large investment in existing functional language implementations; whole platforms are still costly to implement; and there would have to be a convincing economic reason for functional language implementations to converge and interoperate. Fortunately for the author, ILX has served other purposes as well, discussed further in the conclusion. Perhaps the most important has been to help identify some minimal changes required to the CLR design and implementation to best support the designs described here.

1.3 The ILX Design Philosophy

Our design constraints for ILX have been as follows:

Compatibility. We maintain full compatibility with CIL. The ILX language includes all existing CIL constructs.

Adequacy for certain languages. The constructs must be adequate to compile the equivalent constructs in ML, O'Caml, Haskell and Mercury. Note that adequacy is more important than minimality – if a "kitchen-sink" approach is required, then so be it. ILX is not a "common language subset", but the language out of which compatible subsets can be specified.

Reasonable efficiency. The design must, in theory, be translatable to existing CIL constructs. The translated code must naturally conform to the basic rules of the .NET CLR, and a verifiable translation must be possible in theory, perhaps by sacrificing some efficiency. Essentially the implementation by translation must be fast enough to mean that compiler writers would prefer to use these constructs rather than finding their own encodings in CIL.

Feasibility of direct support. The constructs must be appropriate to implement directly in some future version of the CLR. Looking to the future, the designs should in principle support a range of optimization strategies.

Non-enforceability of all language properties. We do not try to preserve all high-level language properties in the underlying ILX language. For example, in our present design discriminated unions are always mutable data structures, even if some high-level languages typically optimize on the basis of non-mutability.

Design orthogonality. The design of the different extensions must be orthogonal. Indeed, as we implement the ILX assembler by eliminating the constructs one by one, they also need to be orthogonal for implementation reasons.

2 The .NET Common Language Runtime

A dynamic execution platform (or virtual machine, runtime environment or execution engine) is the combination of a number of software services, usually packaged as a process running on an underlying operating system. The services typically include execution of a bytecode-based portable binary format, a garbage collector and a set of standard libraries. Other services may include Just-In-Time (JIT) compilers, marshalling and distribution support, some kind of code security via encryption, signing and/or code verification checks, support for reflective programming, and developer services such as debugging and profiling APIs. Examples include the O'Caml bytecode interpreter [19], the Java Virtual Machine [20], and Microsoft's .NET CLR [5]. In this paper we focus on the CLR, though we emphasize that our approach to extending an IL could be applied in other situations, and that many of the technical details would carry over.

The CLR "manages" the execution of code. When developing code with a language compiler that targets the CLR, you compile your application or component to code that uses the services described above. CIL code contains "metadata" that makes it "self-describing", so it can be disassembled into a fairly readable form and used by tools, especially to provide a common model for component interaction.

The CLR has been designed explicitly to support multiple programming languages. In particular, it can execute C#, C, C++ and VB.Net code efficiently and faithfully, and the constructs required to achieve this have greatly affected its design. Supporting multiple languages is both a blessing and a curse. Different languages have different execution models, and the design of the CLR must, necessarily, end up looking somewhat like the "union of several different dynamic execution platforms". For example, to efficiently execute C code that assumes a 64-bit processor, but which explicitly performs bit-level operations on pointers, the machine must certainly provide a rich model of integer types and pointer arithmetic. To support VB, instructions that allow a certain kind of "safe" use of pointers (called "byref parameters") are required. To support object-oriented languages, an object model is required, and to support Java-style arrays, co-variance is allowed between array types, even though many languages do not require this. A unified model of several kinds of exceptions is also included. Many languages have been successfully targeted at the CLR, including Mercury, Standard ML, C#, C++ and VB.NET. Any one language typically only needs a small number of the features provided.

The CLR provides numerous other services that are well beyond the scope of this report, e.g. a declarative security model and "assemblies" for managing code packages. One of the primary motivations for extending an existing IL is that we can take advantage of all these features from other languages.

2.1 CIL

Common IL (CIL) is the intermediary language supported by the .NET Execution Engine. CIL is a stack-based language designed to be easy to generate from source code by compilers and other tools. Some of the instructions of CIL are shown in Figure 1.

Figure 2 shows a sample piece of CIL containing an abstract enumeration class and a method that uses this class to adds up all the lengths in a finite sequence of strings. The CIL contains a subset that is strongly typed and can be verified prior to execution. Some aspects of CIL have been formalized by Gordon and Syme [11]. Note that most IL instructions are polymorphic with respect to size, e.g. there is just one `add` instruction. The JIT reconstructs the basic types on the stack in order to determine the machine code to emit.

Primitive Types	Arrays	Classes
Methods	Fields	Construction
Properties	Enumerations	Exceptions
Interfaces	Events	Visibility
Identifiers	Value Classes	

Figure 3: Constructs included in the .NET CLS

Logically speaking, the .NET CLR has a much richer input language than is immediately evident from the CIL instruction set. This is for two reasons: assembly (the .NET notion of a software component), class, interface, "struct" and method definitions form part of the CIL and give a fairly rich object calculus for organizing and expressing data structures. This calculus includes a fairly standard kind of privacy (public, private, protected and assembly scoping) and a selection of novel declarative security attributes. Secondly, some aspects of CLR's behaviour are by rights first-class constructs in the CIL and are thinly disguised within other constructs (e.g. delegate and enumerations) or as calls to class libraries (e.g. dynamic security scopes and threads).

2.2 The CLS

The .NET design already specifies an interoperability standard called the .NET Common Language Specification (CLS), for interoperability between object-based languages such as VC++, Eiffel, C# and VB. The constructs included are shown in Figure 3. Many constructs are in the CLR but not in the CLS. These include finalization, immutability, variable argument functions and optional parameters.

3 The ILX Extensions for First-class Function Values

We now proceed to decribe the extensions to the CIL currently included in ILX. It has long been recognized that "anonymous computations" (i.e. lambda terms or function values) are pervasive in computing. Often they are side-effect free, but anonymous computations with limited side-effects also have many uses. Languages that fail to build in an adequate notion of anonymous computation invariably force the programmer to use contorted techniques in many situations, e.g. the simplest "map" operations will have no natural representation.

Anonymous computations need not be statically typed, but type systems often include a notion of a function type. Functions can sometimes also accept other things as arguments besides values, e.g. type functions accept types as arguments – we consider this case in §5.3.

One of the key goals in this section is to choose a design that allows an efficient implementation, in particular one where each closure is compiled to one method, and *not* one class as is the case in many existing implementations for

`int32`, `int64`, `float32`, `float64`, `native int`, ...	Types for "32-bit integers" etc.
`ldc.i4`, `ldc.r4`, `add`, `sub`, `mul`, `div`, `shl`, `shr`, ...	"Load integer constant", "add" and other arithmetic instructions.
`ldloc`, `stloc`, `ldarg`, `starg`	"Load local", "Store local", etc. Manipulate resources local to a method invocation.
`ldfld`, `stfld`, `ldsfld`, `stsfld`	"Load field", "Store field", "Load static field", etc. Manipulate object and static data.
`ldloca`, `ldarga`, `ldflda`, `ldsflda`, `ldind`, `stind`	"Load local address", "Load argument address" etc. Pointer manipulation.
`call`, `callvirt`, `newobj`	Method call and object creation.
`ldftn`, `calli`	C-style code pointer generation and use.
`castclass`, `isinst`	Access runtime type information.
`box`, `unbox`	Convert in-line values ("structs") to and from heap-allocated objects.

Figure 1: Some Sample CIL Types and Instructions

```
.class abstract Enumerator {
    .method public abstract bool GetNext() { }
    .method public abstract System.Object GetObject() { }
}

.method static int32 MyTotalLengths(Enumerator myEnum) {
    .locals(int32 n, System.String s)
          ldc 0
          stloc n
    loop: ldarg myEnum
          callvirt instance bool Enumerator::GetNext()
          beq done
          ldarg myEnum
          callvirt instance System.Object Enumerator::GetObject()
          castclass System.String
          callvirt instance int32 System.String::GetLength()
          ldloc n
          add
          stloc n
          br loop
    done: ldloc n
          ret
}
```

Figure 2: A Sample Class Definition in CIL

virtual machines. Furthermore we ensure that it is feasible to implement the design using some of the standard multiple entry point tricks from the literature.[2]

3.1 The ILX Design

We first describe the ILX extensions for function types, closures and function values, and then consider some implementation techniques being explored in the current ILX implementation.

The ILX design includes the following aspects.

- n-ary function types. This is a space of function types of the form $\tau_1, ..., \tau_n \rightarrow \tau$, written `(func (`$\tau_1$`, ...,`$\tau_n$`) -->` τ`)`. The number of arguments accepted by each application may be zero or greater.
- Closures classes. Closure classes can be declared to accept one or more groups of multiple arguments, i.e. closures may be declared in "curried" form. Closure classes that accept one empty set of arguments can be declared to have thunking-semantics.
- Function application is performed using the `callfunc` instruction. This applies one or more groups of arguments. The function value appears first on the stack, followed by the argument groups in sequence.
- Closure types. Closure class declarations introduce corresponding closure types. Type annotations can then be used to show that function values are known to belong to particular closure types. An instruction `callclo` to perform a "direct call" to a closure is provided.
- Subtyping rules. These are defined for the new types: function types are subtypes of System.Object, closure types are subtypes of their corresponding function types.[3]
- Runtime typing and reflection rules for the new types. The CIL instructions `castclass` and `isinst` can be used on function types with guaranteed "exact" results. The instruction `castclo` can be used on closure types with "inexact" results (see §3.4 below).
- Instructions `ldenv` and `stcloenv` to access and perform limited mutation on the closure environments. Closure environments may only be mutated in the same block of code where the function value is allocated. If general mutability is required then appropriate fields of the closure environment must be boxed.
- Cross module closure references. Optimizing compilers can reference closures declared in other modules directly (see the Appendix).

None of this is terribly surprising in itself – it embodies a straightforward eval-apply model of function values in the context of CIL. The key aspects are the fact that the design is fully integrated with existing CIL constructs and that it permits a range of implementation options. Most importantly, our design does not commit a translating implementation to realize a closure "class" by a CIL "class" – indeed in one of our implementations each closure corresponds to a single CIL method. We discuss this further below, but the basis for this result is that the information that can be specified in a closure class is very limited, and the reflection semantics of function values are under-defined. Closure classes may *not* contain additional methods, fields, attributes, data, security declarations or any of the other baggage that comes with regular CIL classes.

Figure 4 shows how closures are declared and how a function value is created and called. The declarations that follow the name of the closure are the members of the environment – the order and names of these declarations are irrelevant to the execution semantics. Each closure class must have one `.apply` method.

3.2 Mutating Environments

The semantics of mutating an environment (using `stcloenv`) are undefined, except in the code block where the function value is allocated. This mutation is allowed to permit fix-ups of the environment for mutually recursive functions. If exact mutation semantics are required in other situations, then the closure environment should be a reference to an object that stores the environment – there will then be no need to mutate this reference itself. This allows implementations to copy environments as necessary when building partial applications.

3.3 Thunks

Figure 5 shows a sample ILX program that defines a thunk. The evaluation code is guaranteed to be executed only once and the result memoized. The environment is also "copied-out" to the stack when application occurs to prevent space-leaks. Otherwise the declarations and instructions used are identical to closures.

Classes may subtype zero-arity function types in one particular way, for example:

```
class MyString : (func () --> class MyString) {
  .field public class System.String mydata;
}
```

Function application always returns the object itself for such classes. This is allowed to permit data values to be used where computations are expected. Only values of classes declared via this route are guaranteed to be compatible with their lifted type.

3.4 Closure Types

Closure types indicate that a value is not only known to be a particular function type, but is also known to be an instance of a particular closure class. The only special you can do with such a value is perform an application using `callclo`, which are normally implemented as faster, direct calls. Classes may not subtype closure types.

The instruction `castclo` can be used on a closure type. However, unlike class types, the mapping between `.closure` declarations and runtime closure types need not be 1:1. That is, the ILX may choose to make two `.closure` declarations indistinguishable at runtime. In practice compilers should never emit `castclo` instructions that may fail. If

[2] Note to the referees: I am currently enquiring with some Scheme and Lisp implementors about the best reference to include at this point – it is surprisingly difficult to find a single reference that captures the wealth of experience with implementing functional languages, especially when it comes to tricks such as multiple entry points.

[3] Function types are not currently co/contra-variant due to the limitations this would cause on some implementation techniques.

```
// The two closure classes implement the function types:
// (func (int32, int32) --> int32) and
// (func (int32,float32) --> (func (int32) --> (func (int32,float32) --> int32)))
.closure add() {
  .apply (int32 x,int32 y) --> int32 {
      ldarg x ldarg y add ret
  }
}
.closure add_lots(int32 fv1,float64 fv2) {
  .apply (int32 x1,float32 x2) (int32 x3) (int32 x4,float32 x5) --> int32 {
      ldenv fv1 ldenv fv2 conv.i4 add
      ldarg x1 add ldarg x2 conv.i4 add
      ldarg x3 add ldarg x4 add
      ldarg x5 conv.i4 add
      ret
  }
}

.method static public void main() {
   .locals(int32 result, (func (int32) --> (func (int32,int32) --> int32)) f)
   .entrypoint

   // allocate a function value:
   ldc.i4 10
   ldc.r8 3.1415
   newclo closure add_lots

   // partially apply it:
   ldc.i4 20
   ldc.r4 2.71
   callfunc (int32,float32) --> (func (int32) --> (func (int32,float32) --> int32))
   stloc f

   // apply the remaining arguments:
   ldloc f
   ldc.i4 30 ldc.i4 40 ldc.r4 1.11111
   callfunc (int32) (int32,float32) --> int32
   stloc result

   // print the result:
   ldloc result
   call void class System.Console::WriteLine(int32)
   ret
}
```

Figure 4: Sample ILX Code using First-class Functions

```
// The thunk class implements the function type
// (func () --> int32)
.thunk mythunk (int32 x) {
  .apply () --> int32 {
     ldenv x 17
     ldc.i4 17
     ret
  }
}
.method static public void main() {
   .locals(int32 result, (func () --> int32) f)
   .entrypoint

   // Allocate a thunk closure.
   ldc.i4 10
   newclo closure mythunk
   stloc f

   // Evaluate the thunk.
   ldloc f
   callfunc () --> int32
   stloc result

   // Evaluate again.
   ldloc f
   callfunc () --> int32
   stloc result
   ret
}
```

Figure 5: Sample ILX Code using Thunks

privacy of closure environments is required, then the environment should be wrapped in a class declared private to the containing assembly.

3.5 Function Types and Delegates

The .NET CLR already includes a notion that is not too far from function types and anonymous computations, called delegates. For those unfamiliar with delegates, the guide in Figure 6 may be helpful.

Delegate types are essentially named function types, introduced by special kinds of class declarations, and it is worth considering whether one could unify the two concepts, partly because delegate types are used frequently in the .NET standard libraries. Delegates can be used to implement most of the ILX design described above: closures would simply become delegate objects, and typically the closure will itself be the delegate recipient associated with the delegate object. However, as they stand, delegates are not adequate to use for function types and closures: delegate types are named, not structural; the code for the delegate is tied to the environment; each closure class corresponds to roughly one delegate recipient class; delegate types are named, rather than belonging to a general space of function types; and the costs of delegate construction, invocation and the space-costs of delegate objects appear high on existing CLR implementations, at least when compared with typical functional language implementations.

Nevertheless we plan to experiment with a translation to delegates, perhaps combined with some modifications to the CLR, and to determine the runtime costs involved.

3.6 Implementation Strategies

In this section we consider techniques to implement the above design for function types and closures via translation to (perhaps unverifiable) CIL code. We also consider taargeting Generic CIL, described in [17] and covered further in §5. A direct implementation of the constructs in the CLR is naturally possible but is beyond the scope of this paper.

One of the long-term goals of ILX is to determine the "best" such translation given the overall goals of ILX, and to identify the minimal modifications to the CIL (if any) that are required to support it. We are in the process of completing a spectrum of implementations and comparing their performance under various parameters.

The key aspect of any such implementation is a *closure conversion*. A range of type-preserving closure conversions have been studied in other contexts [21, 1]. One of the key aspects of such a conversion is the use of existential types to model environments, and in our situation we use classes, subclasssing and subtyping (which offer a form of existential typing) as a replacement.

The simplest closure conversion to CIL is as follows:

- Function types are translated to a set of polymorphic abstract base classes `Func0<B>`, `Func1<A,B>`, `Func2<A1,A2,B>` up to some `Func`N.[4] Each of these have an appropriate abstract virtual `apply` method, e.g. `B apply(A)` for `Func1`. We thus piggyback off the system of polymorphism described in §5 and rely on the implementation of that system to handle code-generation, non-uniform instantiations and the insertion of most of the casts needed for verifiability.
- Closures become subclasses of these classes that implement the base class at a particular type, overriding the entrypoint. Other operations map down to CIL object operations in obvious ways.

Such an implementation has been tried before when implementing languages on the JVM and .NET CLR [8, 1, 26], and suffers from an obvious and well-known problem: there will be many, many classes for a typical functional program (our estimates show one closure per line of Haskell code for the GHC standard library). By necessity, the implementation of classes on such systems is always going to be heavyweight, as classes must support reflection semantics. Other problems include the technical difficulties involved with supporting multiple calling conventions for closures under such a scheme.

Our favoured implementation strategy is rather different, and relies on features of the CLR that are not present in systems such as the JVM. The idea is to have one unverifiable implementation module that encapsulates the unsafe tricks we use to implement closures. This module can use the "function-pointer" primitives of CIL (`calli` and `ldftn` – see §2.1) to mimic the implementation strategies of existing functional language implementations – these are verifiable to a degree in CIL, in the sense that typesafe function pointers can generated and be passed to methods that accept them. As a minimum the module must provide:

- a collection of abstract generic types `Func0<B>`, `Func1<A,B>`, `Func2<A1,A2,B>` up to some `Func`N;

[4] Function types of higher arity would be compiled using the given types combined with a product construct, e.g. a parametric `Pair<A,B>` class.

Delegate types.	Named n-ary function types.
Delegate recipient classes.	Closure classes where the data-layout of the closure is tied to the code.
Delegate objects with an associated delegate recipient.	Closures where the environment has been separated out to be another object in the heap.
Multicast delegates.	Closures that sequence several side-affecting function applications.

Figure 6: Delegates v. First-class Functions

- several abstract generic closure classes, defined to be subclasses of the above function types, e.g. `Clo1<A,B,E> : Func1<A,B>` is used to implement closures containing one free variable with an environment of type `E`;
- methods for allocating function values;
- methods for perfoming function applications.

The allocation methods will be generic and have signatures such as

```
Clo1<A,B,E> bake1<A,B,E>
  (E env, method B *(Clo1<A,B,E> env,A arg) code)
```

where `method` τ `*(`τ_1`, ..., `τ_n`)` is the CLR type for a C-style code pointer.

The application methods are again generic and have signatures such as

```
B app1<A,B>(Func1<A,B> f, A arg)
```

These functions will simply tailcall to the code pointer carried in the closure `f` and should normally be inlined.

The ILX operations then map down to these constructs in a straightforward fashion. The actual implementation of the above module has many options: the simplest implementation will only support one entry point and the code pointer for each closure will be stored inline in the function value itself.

Further refinements are possible to this latter implementation technique, in particular to support multiple entry points. These are under development and are beyond the scope of this paper.

4 The ILX Extensions for Discriminated Unions

Discriminated unions are a typesafe way of dividing data into categories. They form an essential part of the implementation of recursive sum/product structures found in ML, Haskell, OCaml, Mercury and many other languages, and thereby help provide a simple, unified way of modeling structures such as lists, trees, records, enumerations, and abstract syntax.

.NET does not support discriminated unions directly, and they must instead be encoded in the object model. After a moment's thought it can be seen that there are many ways of doing this, depending on the particular datatype in question, and upon the encoding scheme desired. For example, even simple enumerations can be encoded as 32 bit integers, or as integers of an appropriate size, or as explicitly unsigned integers. Our aim in this section is to relieve the compiler writer of the burden of deciding on appropriate representations. We do this by extending CIL with constructs to define and manipulate discriminated unions in a way that is completely compatible with existing CIL. This will allow those languages whose compilers generate ILX for their corresponding constructs to interoperate to some degree.

The ILX design for discriminated unions is fairly straightforward, the main questions being ones of implementation and the guarantees that would be given as to how the underlying constructs appear as classes to the C# or CLS programmer. The extensions are as follows. New algebraic types are defined using the ".classunion" directive:

```
.classunion color {
  .alternative RGB(int32,int32,int32)
  .alternative CMY(int32,int32,int32)
  .alternative HSB(int32,int32,int32)
}
```

Each alternative specifies a name and a signature, the latter representing the data fields that objects of that kind posess. Note that data fields need not be given names, i.e. names of data fields are not significant for purposes of binding (linking) or (non-reflective) execution. Constructors with the same name but different signatures are distinct, so some overloading is permitted.

Classunions are much closer to normal CIL classes than closures, because they are not anonymous and we require less flexibility in how they are implemented. They may contain:

- static members;
- instance methods, both virtual and non-virtual;[5]
- custom attributes and security attributes;
- static and instance fields, the instance fields belonging to the overall class and thus to each alternative;

They may also extend other classes. About the only restriction is that classunions are implicitly sealed and may not have explicit layout information, restrictions that could in theory be lifted. Thus our classunions are really "classes with an embedded discriminated union."

We introduce instructions `newdata`, `lddata`, `stdata`, `castdata`, `isdata` and `switchdata` to create and manipulate classunion values. The code in Figure 7 illustrates these. The `switcher` method illustrates the use of `switchdata`. At the branch points the data is left on the stack and, as far as

[5] Presently the virtual methods belong to the classunion itself, and as this is final they are effectively non-virtual. However in the future the design may permit virtual method implementations for each alternative.

```
.method static public void main() {
   .locals(classunion color)
   .entrypoint

   ldc.i4 0
   ldc.i4 255
   ldc.i4 255
   newdata classunion color RGB(int32,int32,int32)
   stloc 0

   ldloc 0
   castdata classunion color,RGB(int32,int32,int32)
   ldc.i4 255
   stdata classunion color,RGB(int32,int32,int32),0

   ldloc 0
   castdata classunion color,RGB(int32,int32,int32)
   lddata classunion color,RGB(int32,int32,int32),0
   pop

   // Create a CMY(0,0,0), call isdata on it, throw the result away
   ldc.i4 1
   newdata classunion color CMY(int32,int32,int32)
   isdata classunion color CMY(int32,int32,int32)
   pop
   ret
}

.method static public void switcher(classunion color x) {
   ldarg x
   switchdata classunion color,
              (RGB(int32,int32,int32),rgb),
              (CMY(int32,int32,int32),cmy)
default: pop
         br lab3
rgb:     ldc.i4 1
         stdata classunion color,RGB(int32,int32,int32),0
         br lab3
cmy:     ldc.i4 2
         stdata classunion color,RGB(int32,int32,int32),0
lab3:    ret
}
```

Figure 7: Sample ILX Code using Discriminated Unions

the type system is concerned, has a type corresponding to the appropriate alternative. These types are not first-class in ILX, and only exist for the purposes of intra-method verification.

4.1 Implementation Strategies

Once again we only consider implementations to CIL or Generic CIL. Given this, the range of implementation strategies for discriminated unions is wide but straightforward: the basic encoding will typically be superclass-subclass based, with one subclass for each alternative. There are obvious improvements on this scheme: enumerations can be encoded as integers; other zero-argument alternatives can be encoded either by `null` or by constant members of the superclass; if there is only one non-zero-argument alternative the superclass can act as that alternative; if non-verifiable code is being produced then it may be wiser to use an integer tag to discriminate between alternatives rather than a runtime type. As is well known, polymorphism makes it difficult to be more ambitious than these kinds of global data-layout optimizations.

The exact choice of a "standard" encoding is outside the scope of this paper – this depends partly on the efficiency required an partly whether the results of the translation should be visible to the C# or other .NET programmers. We plan to present a detailed analysis of implementation options when more data is available.

5 The ILX Extensions for Parametric Polymorphism

Generics, or parametric polymorphism, allow classes, methods and other structures to be parameterized by types. In parallel with the ILX project we have designed support for generics as an extension to CIL and implemented it natively in the Microsoft .NET CLR [17]. In this section we summarize this design and explain its basic interactions with the constructs described so far.

The key features of the PP design are:

Type abstraction. Classes, interfaces, structs and methods can be parameterized by type.

Non-uniform instantiations. All polymorphic structures can be instantiated at both reference and non-reference types, the latter including 32-bit unboxed integers, 64-bit unboxed floats and structs.

Polymorphic inheritance. A polymorphic class can extend a monomorphic superclass, and a monomorphic class can extend a instantiation of a polymorphic superclass.

Constraints by interfaces. Type parameters may be constrained by interface and class types, and interfaces used in this way may include static (i.e. "static-virtual") members.

Exact Runtime Type Semantics. Instantiations of type parameters are not erased at runtime, and thus operations such as `castclass` have "exact" type semantics.

Polymorphic virtual methods. Type applications are supported at indirect call-sites such as virtual methods.

Non-variant. For example, there is no typing relationship between `List<String>` and `List<Object>`.

No higher-kinded polymorphism. Abstraction over type constructors (e.g. abstracting over "List" or "Array") is not permitted.

The new CIL instructions and types are

- `class` *class-name*`<`$\tau_1, ..., \tau_n$`>` – instantiated generic class type.
- `value class` *value-class-name*`<`$\tau_1, ..., \tau_n$`>` – instantiated generic struct type.
- `!`n – a class type variable, numbered from the outermost to the innermost textually.
- `!!`n – a method type variable, numbered from the outermost to the innermost textually.
- `call`, `callvirt`, `ldfld`, `stfld`, `newobj`, `newarr` – these instructions are modified to take specifications of instantiations in addition to their other parameters.
- `ldelem.any`, `stelem.any` – these size-polymorphic array access instructions are added (they are normally ldelem.i4, ldelem.ref etc.)
- `castclass`, `isinst` – the semantics of these instructions are modified for exact runtime type semantics.

5.1 Implementation Strategies

ILX provides several implementations of the system described above by translating to CIL code.[6] It can also optionally emit generic code unchanged to run on a CLR with generics implemented natively. The translating implementations currently provided or planned for ILX are:

- Implementation by erasure to the universal representation `System.Object`, inserting box/unbox [18] instructions where necessary, and with loss of exact runtime type semantics;
- A similar implementation that code-expands with respect to two representations (Object and int32);
- An implementation that uses runtime reflection to generate new classes as needed (this is work in progress).

5.2 Polymorphism and Discriminated Unions

Given the design for Generic CIL described above, we must consider how polymorphism interact with the designs described so far. We consider discriminated unions first. Here the design is simple: discriminated unions may be tagged with generic type parameters:

[6] While the syntax of ILX's polymorphism is identical to that for the Generic CLR, the current implementation does not have identical semantics. In particular, we do not always maintain exact runtime type information. This is considered a bug in the current ILX implementation. We had planned to implement this, but now prefer to use the Generic CLR where possible rather than invest time in the required rather complex encoding for ILX. The current languages targeting ILX are not affected as they do not make use of exact runtime types.

```
.closure identity<T>() {
  .apply (T x) --> T { ldarg x ret }
}
.closure mem_inner_closure<T>(T x) {
  .apply (List<T> list) --> int32 {
     // Loop through ''list'' looking for ''x''
  }
}
.method static void main() {
   // Create an instance of the closure, with T = int32
   ldc.i4 17
   newclo mem_inner_closure<int32>
   ...
   // Assume a List<int32> is pushed on the stack.
   // Now apply the closure. (We could use a direct call:
   //  callclo mem_inner_closure<int32>)
   callfunc (List<int32>) --> int32
}
```

Figure 8: Sample ILX Code for Polymorphic Closures

```
.classunion tree<any T>
  .alternative Tip()
  .alternative Node(tree<T>, tree<T>)
```

As noted in §3.6, polymorphism limits implementation options in some ways when choosing representations for datatypes. The exact runtime type semantics also limit implementations: for the above discriminated union, values of the nullary constructor `Tip` must de discriminable for different `T`, e.g. `Tip<int>` must, in some way, be a different value to `Tip<string>`, as both are compatible with the type `Object` and casting can be used to differentiate one from the other. This problem can be addressed in a number of ways, but the use of the `null` value to represent `Tip` will not be possible.

5.3 Polymorphism, First-class Function Values and Type Functions

Polymorphism is more complex to combine with closures. Some decisions are easy: the first is to permit closure classes to be parameterized by type, as illustrated in Figure 8. These type parameters are effectively the free type variables in the corresponding λ expression.[7] These free type variables must be specified when a closure is created and whenever the closure class is used "directly", e.g. in the `callclo` instruction. These kind of type parameters are *not* specified when using `callfunc`.

The ILX design goes further than this, however, and supports a notion of first-class type function. Given the presence of a system of generics that permits the runtime application of type parameters, it is natural to extend our notion of function value to encompass type functions, i.e. anonymous function values accepting types as arguments. As it happens (and this is no coincidence) our target system of generics [17] includes virtual methods that accept type parameters as arguments: these are precisely type applications at indirect callsites, i.e. at runtime. However that system does not include any types of the form $\forall\alpha.\tau$.

[7] I have found this a useful insight while analysing the kinds of polymorphism supported by systems based on parameterized classes such as C++ templates, GJ, Ada generics and PolyJ [2, 22, 25]

The code in Figure 9 shows an ILX code sample where a type function is created and then used within a method. The extensions for type functions are:

- A new form of types of the form $\forall\alpha.\tau[\alpha]$, written `(forall <any> ...)` and indexing type parameters from the outermost quantifier inward.
- Modifications to the instructions `callfunc` and `callclo`, which may now incorporate type applications. For a single type application `callfunc <ty1> --> ty2`, a value with static type `(forall <any> ty2` must be on the stack, and an object of static substituted type `ty2[ty1]` is returned.
- Subtyping rules. Type functions are always subtypes of `System.Object`. The following subtyping rule also always holds: $\forall\alpha.\tau[\alpha] < \forall\alpha.o$ where o is `System.Object`.
- Exact runtime type semantics for the new types (though currently the ILX implementation does not correctly implement these semantics.)

It is expected that `callfunc` instructions involving type applications are executed rarely, primarily upon class and object initialization. Most languages such as Standard ML and Haskell allow nearly all type applications to be lifted to become free at the "top level", or into module initialization code.

6 Related Work

ILX is one attempt to transfer results that have been described again and again in an academic setting across to a "real-world" context. Two areas of recent work are particularly relevant: type systems for low-level languages (a good summary is in [6]), and the efforts to incoporate algebraic language features into Java, e.g. Pizza, NextGen and Bücki and Weck's work on compund types [23, 4, 3]. In many ways our work is much closer in spirit to the latter, as we accept an existing language as our starting point and are trying to retrofit constructs on top of this. Our constraints have been somewhat different, as we are trying to satisfy the needs of languages such as Standard ML, Haskell and Mercury, rather than the needs of Java programmers, but the recurring problems of finding a design that "feels right" (i.e. has an appropriate set of orthogonal properties) given an existing language has been similar. Both kinds of work share a common requirement to "keep it simple", in our case in order to make sure that different languages compile constructs in compatible ways.

The translation steps performed by our implementation of ILX are strongly reminiscent of those performed by Morrisett et al. when implementing TAL [10]. The also share similarities with the implementation steps in compilers targeting the JVM or .NET CLR, e.g. [1].

There is a growing body of work on interlanguage working. Our work differs from that based on marshalling, COM APIs or FFIs, mainly because we seek language interoperability via language integration rather than marshalling techniques.

7 Conclusions

This paper has presented the design aims for ILX and described the ILX design choices for function types, closures,

```
// The following specifies a closure that accepts a type as its first srgument.
// Objects of this closure have type ∀α. α → α.
.closure id () {
  .apply <any> (!0 x) --> !0  {
     ldstr "Called id once..."
     call void System.Console::WriteLine(class System.String)
     ldarg x
     ret
  }
}

// This method accepts type function as an argument.
// It applies it twice at different types:
.method void go( (forall <any> (func (!0) --> !0)) f) {
   ldarg f ldc.i4 17 callfunc <int32> (!0) --> !0 pop
   ldarg f ldc.i4 "abc" callfunc <class System.String> (!0) --> !0 pop
   ret
}

.method static void main() {
   // Create a value that is a type function:
   newclo class id
   // Pass the type function as an argument:
   call void go((forall <any> (func (!0) --> !0)))
   ret
}
```

Figure 9: Sample ILX Code for Type Functions

thunks, discriminated unions and parametric polymorphism. ILX makes implementing these aspects of a language on the .NET platform simple, by providing these constructs within the context of the existing IL. Furthermore, such a compiler using ILX will automatically produce code that is representationally compatible with other ILX implementations, thus making interlanguage working feasible.

Furthermore ILX takes advantage of features of the .NET execution platform – for example the inclusion of both exact runtime types and type functions only really makes sense on an execution environment that can perform Just-In-Time code specialization.

ILX expects types to be preserved through the compilation process. This can make it difficult to adapt an existing compiler (it was the primary difficulty in targeting the GHC compiler at ILX), and also throws into doubt the suitability of ILX for untyped languages such as Scheme. We are in the process of reconsidering design and implementation options to better support untyped languages, especially, of course, with regard to closures.

Our type system is not higher-kinded, and thus is not quite capable of directly representing either the essence of the ML type system (see [24, 13]) or Haskell's higher-kinded type abstraction. This is a significant problem that is difficult to solve – we have considered supporting higher-kinded abstraction in our system of generics for the CLR but the complexity increase is high for the benefits achieved.

ILX has served other purposes for the author besides those outlined in §1. In particular:

- ILX has allowed MSR Cambridge to design and prototype the design for generics for the .NET CLR at an early stage. It has also allowed us to perform certain controlled performance tests for generics.
- We have used it to prototype other suggested design changes for the .NET CLR.
- We are using it to systematically investigate encodings of constructs described with a view to fixing and standardizing them.
- It has helped us give evidence to CLR teams about where performance should be improved, or else the design changed.

In the long run, one major advantage of using a dynamic execution platform is the opportunity that it gives for advanced optimization strategies, as such a platform can utilize incrementally collected global information to make compilation decisions. For example, a dynamic execution platform can inline function calls across compilation units, something that compilers cannot do unless they can see the code being called at compile time and which in any case tends to break versioning properties of the generated code. There are undoubtedly many potential runtime optimization strategies applicable to the kinds of constructs considered in this paper, and this forms a large area for potential future research.

Many more constructs could be systematically encoded at the level of ILX. Some constructs we have considered adding to ILX in the future are:

- Backtracking;
- Type classes [12];
- Compound types [3];
- Join-style synchronization points;
- Covariant return types and contravariant argument types.

Finally, the implementation techniques described in this paper are still under development and require further, continual investigations as the target .NET platform and its implementations evolve.

Acknowledgements

I am deeply grateful to Nick Benton, Cedric Fournet, Andrew Kennedy, Andy Gordon, Simon Peyton Jones, Claudio Russo, Reuben Thomas and Andrew Tolmach for their help and advice with this work.

References

[1] P. N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*. ACM, October 1998.

[3] M. Büchi and W. Weck. Java needs compound types. Technical Report 182, Turku Centre for Computer Science, June 1998.

[4] R. Cartwright and G. L. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.

[5] Microsoft Corporation. The .NET Common Language Runtime. See website at `http://msdn.microsoft.com/net/`.

[6] K. Crary and G. Morrisett. Type structure for low-level programming langauges. In *International Colloquium on Automata, Languages, and Programming*, pages 40–54. Springer-Verlag, July 1999.

[7] R. DeLine and M. Fähndrich. The Vault Project. See website at `http://research.microsoft.com/projects/Vault`.

[8] M. DePristo and P. Surana. The Hot-Dog Scheme compiler. See website at `http://rover.cs.nwu.edu/~scheme`.

[9] T. Dowd and F. Henderson. The .NET Common Language Runtime Mercury Compiler. See website at `http://www.cs.mu.oz.au/research/mercury/`.

[10] G.Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 527–568. ACM, January 1998.

[11] A. Gordon and D. Syme. Typing a multi-language intermediate code. In *27th Annual ACM Symposium on Principles of Programming Languages*, January 2001.

[12] C.V. Hall, K. Hammond, S.L. Peyton Jones, and P.L. Wadler. Type classes in Haskell. In *European Symposium On Programming*, number 788 in LNCS, pages 241—256. Springer Verlag, April 1994.

[13] R. Harper, J.C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 341–354. ACM, 1990.

[14] Anders Hejlsberg and Scott Wiltamuth. C# language reference. See `http://msdn.microsoft.com/vstudio/`.

[15] S.L. Peyton Jones and many others. The Glasgow Haskell Compiler. See website at `http://www.haskell.org/ghc`.

[16] S.L. Peyton Jones, P.L. Wadler, and many others. The Haskell 98 report. See website at `http://www.haskell.org`.

[17] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *ACM-SIGPLAN 2001 Conference on Programming Language Design and Implementation*. ACM, June 2001. To be published.

[18] X. Leroy. Unboxed objects and polymorphic typing. In *19th Annual ACM Symposium on Principles of Programming Languages*, pages 177–188. ACM, 1992.

[19] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. The Objective Caml system users' manual. See website at `http://caml.inria.fr`.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[21] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *23rd Annual ACM Symposium on Principles of Programming Languages*, pages 271–283. ACM, January 1996.

[22] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *24th Annual ACM Symposium on Principles of Programming Languages*, pages 132–145. ACM, January 1997.

[23] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Pizza into Java: Translating theory into practice. In *24th Annual ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, January 1997.

[24] C.V. Russo. Non-Dependent Types for Standard ML Modules. In *1999 International Conference on Principles and Practice of Declarative Programming*. September 1999.

[25] Robert A. Duff (Eds.) S. Tucker Taft. *Ada 95 Reference Manual: Language and Standard Libraries*, volume 1246. Springer-Verlag, 1997. International Standard ISO/IEC 8652:1995(E).

[26] G. Stolpmann. The JavaCaml bytecode interpreter. See website at `http://www.ocaml-programming.de/javacaml`.

[27] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8:367–412, July 1998.

A The ILX Assembler and SDK

This appendix describes some aspects of the ILX assembler and the current ILX SDK. These do not form part of the core paper as one might equally implement the extensions directly in the dynamic execution platform environment.

A.1 Extra Information Required for a Context-Free Translation

The ILX assembler (ILX2IL) is at somewhat of a disadvantage in contrast to a direct implementation. For example, in the context of CIL, a JIT compiler can directly resolve references to named classes, for example to determine the number of fields in a value class. However the ILX assembler can only access the information in the input file itself, and, in keeping with typical assembly languages, the importing of context information such as header files is not allowed.

To ensure that we can translate constructs we require two additional sets of declarations:

- A declaration for each closure (or thunk) defined in an external module and used in a `newobj`, `callclo` or `castclo` instruction.
- A declaration for each discriminated union defined in an external module and used in one of the new instructions from §4.

These declarations are required to support cross-module mutual recursion and referencing. As an example, the following declaration defines a "reference to an externally defined classunion type":

```
.classunion import List<any T> {
  .alternative Nil()
  .alternative Cons(T, class List<T>)
}
```

Note that the amount of information given in these declarations effectively limits the range of possible translation schemes. For example, the signatures for each alternative in a discriminated unions must be specified but discriminated unions used within these signatures need *not* be specified – i.e. the implementation need not have access to the transitive closure of all datatype declarations. This means that representation schemes for discriminated unions that "flatten" such types together cannot be implemented.

A.2 The ILX SDK

The ILX SDK is made up of the following tools:

ilx2il.exe Converts ILX assembly language (`.ilx`) files to CIL (`.il`) files. Various options control the conversion.

ilxasm.exe Converts ILX assembly language files directly to .NET PE binary files (n.b. not implemented in the current release).

ilxverify.exe Verifies ILX assembly language files using the typing rules of ILX and a typing environment given on the command line.

ilxvalid.exe Performs a weaker set of validation checks on ILX assembly language files.

mkvlb.exe Builds a type environment library to pass to the `ilxverify.exe` and `ilxvalid.exe` tools.

msilxlib.dll The ILX runtime support library.