Design and Implementation of Generics for the .NET Common Language Runtime

Andrew Kennedy Don Syme

Microsoft Research, Cambridge, U.K.
{akenn,dsyme}@microsoft.com

Abstract

The Microsoft .NET Common Language Runtime provides a shared type system, intermediate language and dynamic execution environment for the implementation and inter-operation of multiple source languages. In this paper we extend it with direct support for parametric polymorphism (also known as generics), describing the design through examples written in an extended version of the C# programming language, and explaining aspects of implementation by reference to a prototype extension to the runtime.

Our design is very expressive, supporting parameterized types, polymorphic static, instance and virtual methods, "F-bounded" type parameters, instantiation at pointer and value types, polymorphic recursion, and exact run-time types. The implementation takes advantage of the dynamic nature of the runtime, performing just-in-time type specialization, representation-based code sharing and novel techniques for efficient creation and use of run-time types.

Early performance results are encouraging and suggest that programmers will not need to pay an overhead for using generics, achieving performance almost matching hand-specialized code.

1 Introduction

Parametric polymorphism is a well-established programming language feature whose advantages over dynamic approaches to generic programming are well-understood: safety (more bugs caught at compile time), expressivity (more invariants expressed in type signatures), clarity (fewer explicit conversions between data types), and efficiency (no need for run-time type checks).

Recently there has been a shift away from the traditional compile, link and run model of programming towards a more dynamic approach in which the division between compile-time and run-time becomes blurred. The two most significant examples of this trend are the Java Virtual Machine [11] and, more recently, the Common Language Runtime (CLR for short) introduced by Microsoft in its .NET initiative [1].

The CLR has the ambitious aim of providing a *common* type system and intermediate language for executing programs written in a variety of languages, and for facilitating inter-operability between those languages. It relieves compiler writers of the burden of dealing with low-level machine-specific details, and relieves programmers of the burden of describing the data marshalling (typically through an interface definition language, or IDL) that is necessary for language interoperation.

This paper describes the design and implementation of support for parametric polymorphism in the CLR. In its initial release, the CLR has no support for polymorphism, an omission shared by the JVM. Of course, it is always possible to "compile away" polymorphism by translation, as has been demonstrated in a number of extensions to Java [14, 4, 6, 13, 2, 16] that require no change to the JVM, and in compilers for polymorphic languages that target the JVM or CLR (MLj [3], Haskell, Eiffel, Mercury). However, such systems inevitably suffer drawbacks of some kind, whether through source language restrictions (disallowing primitive type instantiations to enable a simple erasure-based translation, as in GJ and NextGen), unusual semantics (as in the "raw type" casting semantics of GJ), the absence of separate compilation (monomorphizing the whole program, as in MLj), complicated compilation strategies (as in NextGen), or performance penalties (for primitive type instantiations in PolyJ and Pizza). The lesson in each case appears to be that if the virtual machine does not support polymorphism, the end result will suffer.

The system of polymorphism we have chosen to support is very expressive, and, in particular, supports instantiations at reference and value types, in conjunction with exact runtime types. These together mean that the semantics of polymorphism in a language such as C# can be very much "as expected", and can be explained as a relatively modest and orthogonal extension to existing features. We have found the virtual machine layer an appropriate place to support this functionality, precisely because it is very difficult to implement this combination of features by compilation alone. To our knowledge, no previous attempt has been made to design and implement such a mechanism as part of the infrastructure provided by a virtual machine. Furthermore, ours is the first design and implementation of polymorphism to combine exact run-time types, dynamic linking, shared code and code specialization for non-uniform instantiations, whether in a virtual machine or not.

1.1 What is the CLR?

The .NET Common Language Runtime consists of a typed, stack-based intermediate language (IL), an Execution Engine (EE) which executes IL and provides a variety of runtime services (storage management, debugging, profiling, security, etc.), and a set of shared libraries (.NET Frameworks). The CLR has been successfully targeted by a variety of source languages, including C#, Visual Basic, C++, Eiffel, Cobol, Standard ML, Mercury, Scheme and Haskell.

The primary focus of the CLR is object-oriented languages, and this is reflected in the type system, the core of which is the definition of classes in a single-inheritance hierarchy together with Java-style interfaces. Also supported are a collection of primitive types, arrays of specified dimension, structs (structured data that is not boxed, *i.e.* stored in-line), and safe pointer types for implementing call-by-reference and other indirection-based tricks.

Memory safety enforced by types is an important part of the security model of the CLR, and a specified subset of the type system and of IL programs can be guaranteed typesafe by verification rules that are implemented in the runtime. However, in order to support unsafe languages like C++, the instruction set has a well-defined interpretation independent of static checking, and certain types (C-style pointers) and operations (block copy) are never verifiable.

IL is not intended to be interpreted; instead, a variety of native code compilation strategies are supported. Frequently-used libraries such as the base class library and GUI frameworks are precompiled to native code in order to reduce start-up times. User code is typically loaded and compiled on demand by the runtime.

1.2 Summary of the Design

A summary of the features of our design is as follows:

1. Polymorphic declarations. Classes, interfaces, structs, and methods can each be parameterized on types.

2. Runtime types. All objects carry "exact" runtime type information, so one can, for example, distinguish a `List<string>` from a `List<Object>` at runtime, by looking at the runtime type associated with an object.

3. Unrestricted instantiations. Parameterized types and polymorphic methods may be instantiated at types which have non-uniform representations, *e.g.* `List<int>`, `List<long>` and `List<double>`. Moreover, our implementation does not introduce expensive box and unbox coercions.

4. Bounded polymorphism. Type parameters may be bounded by a class or interface with possible recursive reference to type parameters ("F-bounded" polymorphism [5]).

5. Polymorphic inheritance. The superclass and implemented interfaces of a class or interface can all be instantiated types.

6. Polymorphic recursion. Instance methods on parameterized types can be invoked recursively on instantiations different to that of the receiver; likewise, polymorphic methods can be invoked recursively at new instantiations.

7. Polymorphic virtual methods. We allow polymorphic methods to be overridden in subclasses and specified in interfaces and abstract classes. The implementation of polymorphic virtual methods is not covered here and will be described in detail in a later paper.

What are the ramifications of our design choices? Certainly, given these extensions to the CLR, and assuming an existing CLR compiler, it is a relatively simple matter to extend a "regular" class-based language such as C#, Oberon, Java or VisualBasic.NET with the ability to define polymorphic code. Given the complexity of compiling polymorphism efficiently, this is already a great win.

We wanted our design to support the polymorphic constructs of as wide a variety of source languages as possible. Of course, attempting to support the diverse mechanisms of the ML family, Haskell, Ada, Modula-3, C++, and Eiffel leads to (a) a lot of features, and (b) tensions between those features. In the end, it was necessary to make certain compromises. We do not currently support the higher-order types and kinds that feature in Haskell and in encodings of the SML and Caml module systems, nor the type class mechanism found in Haskell and Mercury. Neither do we support Eiffel's (type-unsafe) covariant subtyping on type constructors, though we are considering a type-safe variance design for the future. Finally, we do not attempt to support C++ templates in full.

Despite these limitations, the mechanisms provided are sufficient for many languages. The role of the type system in the CLR is not just to provide runtime support – it is also to facilitate and encourage language integration, *i.e.* the treatment of certain constructs in compatible ways by different programming languages. Interoperability gives a strong motivation for implementing objects, classes, interfaces and calling conventions in compatible ways. The same argument applies to polymorphism and other language features: by sharing implementation infrastructure, unnecessary incompatibilities between languages and compilers can be eliminated, and future languages are encouraged to adopt designs that are compatible with others, at least to a certain degree. We have chosen a design that removes many of the restrictions previously associated with polymorphism, in particular with respect to how various language features interact. For example, allowing arbitrary type instantiations removes a restriction found in many languages.

Finally, one by-product of adding parameterized types to the CLR is that many language features not currently supported as primitives become easier to encode. For example, n-ary product types can be supported simply by defining a series of parameterized types `Prod2`, `Prod3`, etc.

1.3 Summary of the Implementation

Almost all previous implementation techniques for parametric polymorphism have assumed the traditional compile, link and run model of programming. Our implementation, on the other hand, takes advantage of the dynamic loading and code generation capabilities of the CLR. Its main features are as follows:

1. "Just-in-time" type specialization. Instantiations of parameterized classes are loaded dynamically and the code for their methods is generated on demand.

2. Code and representation sharing. Where possible, compiled code and data representations are shared between different instantiations.

3. No boxing. Due to type specialization the implementation never needs to box values of primitive type.

4. Efficient support of run-time types. The implementation makes use of a number of novel techniques to provide operations on run-time types that are efficient in the presence of code sharing and with minimal overhead for programs that make no use of them.

2 Polymorphism in Generic C#

In this section we show how our support for parametric polymorphism in the CLR allows a generics mechanism to be added to the language C# with relative ease.

C# will be new to most readers, but as it is by design a derivative of C++ it should be straightforward to grasp. The left side of Figure 1 presents an example C# program that is a typical use of the generic "design pattern" that programmers employ to code around the absence of parametric polymorphism in the language. The type `object` is the top of the class hierarchy and hence serves as a polymorphic representation. In order to use such a class with primitive element types such as integer, however, it is necessary to *box* the

Object-based stack

```
class Stack {
  private object[] store;
  private int size;
  public Stack()
    store=new object[10]; size=0;
  }
  public void Push(object x) {
    if (size>=store.Size) {
      object[] tmp = new object[size*2];
      Array.Copy(store,tmp,size);
      store = tmp;
    }
    store[size++] = x;
  }
  public object Pop() {
    return store[--size];
  }
  public static void Main() {
    Stack x = new Stack();
    x.Push(17);
    Console.WriteLine((int) x.Pop() == 17);
  }
}
```

Generic stack

```
class Stack<T> {
  private T[] store;
  private int size;
  public Stack()
    store=new T[10]; size=0;
  }
  public void Push(T x) {
    if (size>=store.Size) {
      T[] tmp = new T[size*2];
      Array.Copy(store,tmp,size);
      store = tmp;
    }
    store[size++] = x;
  }
  public T Pop() {
    return store[--size];
  }
  public static void Main() {
    Stack<int> x = new Stack<int>();
    x.Push(17);
    Console.WriteLine(x.Pop() == 17);
  }
}
```

Figure 1: C# and Generic C# Stack implementations

values. C# inserts box coercions automatically (as in `x.Push(17)`) and requires the programmer to write unbox coercions explicitly (as in `(int) x.Pop()`). Of course, the latter can fail at run-time.

2.1 Using parameterized types

For the average user, polymorphism provides nothing more than an expanded set of type constructors, along with some polymorphic static methods to help manipulate them. For example, a class-browsing tool might show a parameterized version of our example Stack class as:

```
class Stack<T> {
  Stack();              // constructor
  void Push(T);         // instance methods
  T Pop();
}
```

The user now has access to a new family of types which include `Stack<int>`, `Stack<Stack<int>>` and `Stack<string>`. He can write code using these types, such as the following fragment of an arithmetic evaluator:

```
enum op { Add, Neg };
Stack<int> s = new Stack<int>();
...
switch (op) {
  case Add : s.Push(s.Pop() + s.Pop()); break;
  case Neg : s.Push(- s.Pop()); break;
}
```

With the vanilla C# implementation of Figure 1 he would have written:

```
Stack s = new Stack();
...
switch (op) {
  case Add :
    s.Push((int) s.Pop() + (int) s.Pop()); break;
  case Neg :
    s.Push(- (int) s.Pop()); break;
}
```

The programmer can therefore replace the casts at every use of `s` with some extra annotations at the point where `s` is created. The implementation of the arithmetic evaluator using the parameterized Stack class will also be much more efficient, as the integers will not be boxed and unboxed.

2.2 Exact runtime types

The type system of the CLR is not entirely static as it supports run-time type tests, checked coercions and reflection capabilities. This entails maintaining exact type information in objects, a feature that we wished to preserve in our design for polymorphism. Thus each object carries with it full type information, including the type parameters of parameterized types.

In the following example, this ensures that the third line raises an `InvalidCastException`:

```
Object obj = new Stack<int>();
Stack<int> s2 = (Stack<int>) obj; // succeeds
Stack<string> s3 = (Stack<string>) obj; // exception
```

Exact runtime types are primarily useful for reflection, type-safe serialization and for interacting with components that do not, for whatever reason, use fully exact types (*e.g.* use type `Object`):

```
// Read some serialized form of an object:
Object s = ReadFromStream();
// Check "s" is a Stack<int>:
Stack<int> s2 = (Stack<int>) s;
```

2.3 Using polymorphic methods

Polymorphic methods take type parameters in addition to normal parameters. Typically such methods will be associated with some

```
interface IComparer<T> {
  int Compare(T x, T y);
}
interface ISet<T> {
  bool Contains(T x);
  void Add(T x);
  void Remove(T x);
}
class ArraySet<T> : ISet<T> {
  private T[] items;
  private int size;
  private IComparer<T> c;
  public ArraySet(IComparer<T> _c) {
    items = new T[100]; size = 0; c = _c;
  }
  public bool Contains(T x) {
    for (int i = 0; i < size; i++)
      if (c.Compare(x,items[i]) == 0) return true;
    return false;
  }
  public void Add(T x) { ... }
  public void Remove(T x) { ... }
}
```

Figure 2: A parameterized interface and implementation

parameterized class or built-in type constructor such as arrays, as in the following example:

```
class Array { ...
  static void Reverse<T>(T[]);
  static T[] Slice<T>(T[], int ix, int n);
}
```

Here `Reverse` and `Slice` are polymorphic static methods defined within the class `Array`, which in the CLR is a super-type of all built-in array types and is thus a convenient place to locate operations common to all arrays. The methods can be called as follows:

```
int[] arr = new int[100];
for (int i = 0; i<100; i++) arr[i]= 100-i;
int[] arr2 = Array.Slice<int>(arr, 10, 80);
Array.Reverse<int>(arr2);
```

The type parameters (in this case, `<int>`) can be inferred in most cases arising in practice [4], allowing us here to write the more concise `Array.Reverse(arr)`.

2.4 Defining parameterized types

The previous examples have shown the use of parameterized types and methods, though not their declaration. We have presented this first because in a multi-language framework not all languages need support polymorphic declarations – for example, Scheme or Visual Basic might simply allow the use of parameterized types and polymorphic methods defined in other languages. However, Generic C# does allow their definition, as we now illustrate.

We begin with parameterized classes. We can now complete a Generic C# definition of `Stack`, shown on the right of Figure 1 for easy comparison. The type parameters of a parameterized class can appear in any *instance* declaration: here, in the type of the private field `store` and in the type signatures and bodies of the instance methods `Push` and `Pop` and constructor `Stack()`.

C# supports the notion of an *interface* which gives a name to a set of methods that can be implemented by many different classes. Figure 2 presents two examples of parameterized interfaces and a parameterized class that implements one of them. A class can also implement an interface at a single instantiation: for example, `CharSet : ISet<char>` might use a specialized bit-vector representation for sets of characters.

```
class Array { ...
  static T[] Slice<T>(T[] arr, int ix, int n) {
    T[] arr2 = new T[n];
    for (int i = 0; i < n; i++)
      arr2[i] = arr[ix+i];
    return arr2;
} }
class ArraySet<T> { ...
  ArraySet<Pair<T,U>> Times<U>(ArraySet<U> that) {
    ArraySet<Pair<T,U>> r = new ArraySet<Pair<T,U>>();
    for (int i = 0; i < this.size; i++)
     for (int j = 0; j < that.size; j++)
      r.Add(new Pair<T,U>(this.items[i],that.items[j]));
    return r;
} }
```

Figure 3: Polymorphic methods

Also supported are user-defined "struct" types, *i.e.* values represented as inline sequences of bits rather than allocated as objects on the heap. A parameterized struct simply defines a family of struct types:

```
struct Pair<T,U> {
  public T fst; public U snd;
  public Pair(T t, U u) {
    fst = t; snd = u;
  }
}
```

Finally, C# supports a notion of first-class methods, called *delegates*, and these too can be parameterized on types in our extension. They introduce no new challenges to the underlying CLR execution mechanisms and will not be discussed further here.

2.5 Defining polymorphic methods

A polymorphic method declaration defines a method that takes type parameters in addition to its normal value parameters. Figure 3 gives the definition of the `Slice` method used earlier.

It is also possible to define polymorphic *instance* methods that make use of type parameters from the class as well as from the method, as with the cartesian product method `Times` shown here.

3 Support for Polymorphism in IL

The intermediate language of the CLR, called IL for short, is best introduced through an example. The left of Figure 4 shows the IL for the non-parametric Stack implementation of Figure 1. It should be apparent that there is a direct correspondence between C# and IL: the code has been linearized, with the stack used to pass arguments to methods and for expression evaluation. Argument 0 is reserved for the `this` object, with the remainder numbered from 1 upwards. Field and method access instructions are annotated with explicit, fully qualified field references and method references. A call to the constructor for the class `Object` has been inserted at the start of the constructor for the `Stack` class, and types are described slightly more explicitly, *e.g.* `class System.Object` instead of the C# `object`. Finally, `box` and `unbox` instructions have been inserted to convert back and forth between the primitive `int` type and `Object`.

Object-based stack

```
.class Stack {
  .field private class System.Object[] store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr System.Object
    stfld class System.Object[] Stack::store
    ldarg.0
    ldc.i4 0
    stfld int32 Stack::size
    ret
  }
  .method public void Push(class System.Object x) {
    .maxstack 4
    .locals (class System.Object[], int32)
        ⋮
    ldarg.0
    ldfld class System.Object[] Stack::store
    ldarg.0
    dup
    ldfld int32 Stack::size
    dup
    stloc.1
    ldc.i4 1
    add
    stfld int32 Stack::size
    ldloc.1
    ldarg.1
    stelem.ref
    ret
  }
  .method public class System.Object Pop() {
    .maxstack 4
    ldarg.0
    ldfld class System.Object[] Stack::store
    ldarg.0
    dup
    ldfld int32 Stack::size
    ldc.i4 1
    sub
    dup
    stfld int32 Stack::size
    ldelem.ref
    ret
  }
  .method public static void Main() {
    .entrypoint
    .maxstack 3
    .locals (class Stack)
    newobj void Stack::.ctor()
    stloc.0
    ldloc.0
    ldc.i4 17
    box System.Int32
    call instance void Stack::Push(class System.Object)
    ldloc.0
    call instance class System.Object Stack::Pop()
    unbox System.Int32
    ldind.i4
    ldc.i4 17
    ceq
    call void System.Console::WriteLine(bool)
    ret
} }
```

Generic stack

```
.class Stack<T> {
  .field private !0[] store
  .field private int32 size
  .method public void .ctor() {
    ldarg.0
    call void System.Object::.ctor()
    ldarg.0
    ldc.i4 10
    newarr !0
    stfld !0[] Stack<!0>::store
    ldarg.0
    ldc.i4 0
    stfld int32 Stack<!0>::size
    ret
  }
  .method public void Push(!0 x) {
    .maxstack  4
    .locals (!0[], int32)
        ⋮
    ldarg.0
    ldfld !0[] Stack<!0>::store
    ldarg.0
    dup
    ldfld int32 Stack<!0>::size
    dup
    stloc.1
    ldc.i4 1
    add
    stfld int32 Stack<!0>::size
    ldloc.1
    ldarg.1
    stelem.any !0
    ret
  }
  .method public !0 Pop() {
    .maxstack 4
    ldarg.0
    ldfld !0[] Stack<!0>::store
    ldarg.0
    dup
    ldfld int32 Stack<!0>::size
    ldc.i4 1
    sub
    dup
    stfld int32 Stack<!0>::size
    ldelem.any !0
    ret
  }
  .method public static void Main() {
    .entrypoint
    .maxstack 3
    .locals (class Stack<int32>)
    newobj void Stack<int32>::.ctor()
    stloc.0
    ldloc.0
    ldc.i4 17

    call instance void Stack<int32>::Push(!0)
    ldloc.0
    call instance !0 Stack<int32>::Pop()


    ldc.i4 17
    ceq
    call void System.Console::WriteLine(bool)
    ret
} }
```

Figure 4: The IL for Stack and Generic Stack

The right of Figure 4 shows the IL for the parametric Stack implementation on the right of Figure 1. For comparison with the non-generic IL the differences are underlined. In brief, our changes to IL involved (a) adding some new types to the IL type system, (b) introducing polymorphic forms of the IL declarations for classes, interfaces, structs and methods, along with ways of referencing them, and (c) specifying some new instructions and generalizations of existing instructions. We begin with the instruction set.

3.1 Polymorphism in instructions

Observe from the left side of Figure 4 that:

- Some IL instructions are implicitly generic in the sense that they work over many types. For example, `ldarg.1` (in `Push`) loads the first argument to a method onto the stack. The JIT compiler determines types automatically and generates code appropriate to the type. Contrast this with the JVM, which has instruction variants for different types (*e.g.* `iload` for 32-bit integers and `aload` for pointers).
- Other IL instructions are generic (there's only one variant) but are followed by further information. This is required by the verifier, for overloading resolution, and sometimes for code generation. Examples include `ldfld` for field access, and `newarr` for array creation.
- A small number of IL instructions do come in different variants for different types. Here we see the use of `ldelem.ref` and `stelem.ref` for assignment to arrays of object types. Separate instructions must be used for primitive types, for example, `ldelem.i4` and `stelem.i4` for 32-bit signed integer arrays.

Now compare the polymorphic IL on the right of Figure 4.

- The generic, type-less instructions remain the same.
- The annotated generic instructions have types that involve `T` and `Stack<T>` instead of `System.Object` and `Stack`. Notice how type parameters are referenced by number.
- Two new generic instructions have been used for array access and update: `ldelem.any` and `stelem.any`.

Two instructions deserve special attention: `box` and a new instruction `unbox.val`. The `box` instruction is followed by a value type τ and, given a value of this type on the stack, boxes it to produce a heap-allocated value of type `Object`. We generalize this instruction to accept reference types in which case the instruction acts as a no-op. We introduce a new instruction `unbox.val` which performs the converse operation including a runtime type-check. These refinements to boxing are particularly useful when interfacing to code that uses the `Object` idiom for generic programming, as a value of type `T` can safely be converted to and from `Object`.

Finally, we also generalize some existing instructions that are currently limited to work over only non-reference types. For example, the instructions that manipulate pointers to values (`ldobj`, `stobj`, `cpobj` and `initobj`) are generalized to accept pointers to references and pointers to values of variable type.

3.2 Polymorphic forms of declarations

We extend IL class declarations to include named formal type parameters. The names are optional and are only for use by compilers and other tools. The `extends` and `implements` clauses of class definitions are extended so that they can specify instantiated types.

Interface, structure and method definitions are extended in a similar way. At the level of IL, the signature of a polymorphic method declaration looks much the same as in Generic C#. Here is a simple example:

```
.method public static void Reverse<T>(!!0[]) { ... }
```

We distinguish between class and method type variables, the latter being written in IL assembly language as `!!`n.

3.3 New types

We add three new ways of forming types to those supported by the CLR:

1. Instantiated types, formed by specifying a parameterized type name (class, interface or struct) and a sequence of type specifications for the type parameters.
2. Class type variables, numbered from left-to-right in the relevant parameterized class declaration.
3. Method type variables, numbered from left-to-right in the relevant polymorphic method declaration.

Class type variables can be used as types within any instance declaration of a class. This includes the type signatures of instance fields, and in the argument types, local variable types and instructions of instance methods within the parameterized class. They may also be used in the specification of the superclass and implemented interfaces of the parameterized class. Method type parameters can appear anywhere in the signature and body of a polymorphic method.

3.4 Field and method references

Many IL instructions must refer to classes, interfaces, structs, fields and methods. When instructions such as `ldfld` and `callvirt` refer to fields and methods in parameterized classes, we insist that the type instantiation of the class is specified. The signature (field type or argument and result types for methods) must be exactly that of the definition and hence include formal type parameters. The actual types can then be obtained by substituting through with the instantiation. This use of formal signatures may appear surprising, but it allows the execution engine to resolve field and method references more quickly and to discriminate between certain signatures that would become equal after instantiation.

References to polymorphic methods follow a similar pattern. An invocation of a polymorphic method is shown below:

```
ldloc arr
call void Array::Reverse<int32>(!!0[])
```

Again, the full type instantiation is given, this time after the name of the method, so both a class and method type instantiation can be specified. The types of the arguments and result again must match the definition and typically contain formal method type parameters. The actual types can then be obtained by substituting through by the method and class type instantiations.

3.5 Restrictions

There are some restrictions:

- `.class Foo<T> extends !0` is not allowed, *i.e.* naked type variables may not be used to specify the superclass or implemented interfaces of a class. It is not possible to determine the methods of such a class at the point of definition of such a class, a property that is both undesirable for programming (whether a method was overridden or inherited could depend on the instantiation of the class) and difficult to implement (a conventional vtable cannot be created when the class is loaded). Constraints on type parameters ("where clauses") could provide a more principled solution and this is under consideration for a future extension.

- An instruction such as `newobj void !0::.ctor()` is outlawed, as is `call void !0::myMethod()`. Again, in the absence of any other information about the type parameter, it is not possible to check at the point of definition of the enclosing class that the class represented by `!0` has the appropriate constructor or static method.

- Class type parameters may not be used in static declarations. For static methods, there is a workaround: simply re-parameterize the method on all the class type parameters. For fields, we are considering "per-instantiation" static fields as a future extension.

- A class is not permitted to implement a parameterized interface at more than one instantiation. Aside from some tricky design choices over resolving ambiguity, currently it is difficult to implement this feature without impacting the performance of all invocations of interface methods. Again, this feature is under consideration as a possible extension.

4 Implementation

The implementation of parametric polymorphism in programming languages has traditionally followed one of two routes:

- Representation and code specialization. Each distinct instantiation of a polymorphic declaration gives rise to data representation and code specific to that instantiation. For example, C++ templates are typically specialized at link-time. Alternatively, polymorphic declarations can be specialized with respect to *representation* rather than source language type [3]. The advantage of specialization is performance, and the relative ease of implementing of a richer feature set; the drawbacks are code explosion, lack of true separate compilation and the lack of dynamic linking.

- Representation and code sharing. A single representation is used for all instantiations of a parameterized type, and polymorphic code is compiled just once. Typically it is a pointer that is the single representation. This is achieved either by restricting instantiations to the pointer types of the source language (GJ, NextGen, Eiffel, Modula-3), by boxing all non-pointer values regardless of whether they are used polymorphically or not (Haskell) or by using a tagged representation scheme that allows some unboxed values to be manipulated polymorphically (most implementations of ML). Clearly there are benefits in code size (although extra box and unbox operations are required) but performance suffers.

Recent research has attempted to reduce the cost of using uniform representations through more sophisticated boxing strategies [10] and run-time analysis of types [9].

In our CLR implementation, we have the great advantage over conventional native-code compilers that loading and compilation is performed on demand. This means we can choose to mix-and-match specialization and sharing. In fact, we could throw in a bit of boxing too (to share more code) but have so far chosen not to do this on grounds of simplicity and performance.

4.1 Specialization and sharing

Our scheme runs roughly as follows:

- When the runtime requires a particular instantiation of a parameterized class, the loader checks to see if the instantiation is compatible with any that it has seen before; if not, then a field layout is determined and new vtable is created, to be shared between all compatible instantiations. The items in this vtable are entry stubs for the methods of the class. When these stubs are later invoked, they will generate ("just-in-time") code to be shared for all compatible instantiations.

- When compiling the invocation of a (non-virtual) polymorphic method at a particular instantiation, we first check to see if we have compiled such a call before for some compatible instantiation; if not, then an entry stub is generated, which will in turn generate code to be shared for all compatible instantiations.

Two instantiations are *compatible* if for any parameterized class its compilation at these instantiations gives rise to identical code and other execution structures (*e.g.* field layout and GC tables), apart from the dictionaries described below in Section 4.4. In particular, all reference types are compatible with each other, because the loader and JIT compiler make no distinction for the purposes of field layout or code generation. On the implementation for the Intel x86, at least, primitive types are mutually incompatible, even if they have the same size (floats and ints have different parameter passing conventions). That leaves user-defined struct types, which are compatible if their layout is the same with respect to garbage collection *i.e.* they share the same pattern of traced pointers.

This dynamic approach to specialization has advantages over a static approach: some polymorphism simply cannot be specialized statically (polymorphic recursion, first-class polymorphism), and lazy specialization avoids wasting time and space in generating specialized code that never gets executed. However, not seeing the whole program has one drawback: we do not know ahead of time the full set of instantiations of a polymorphic definition. It turns out that if we know that the code for a particular instantiation will not be shared with any other instantiation then we can sometimes generate slightly better code (see §4.4). At present, we use a global scheme, generating unshared code for primitive instantiations and possibly-shared code for the rest.

The greatest challenge has been to support exact run-time types and at the same time share representations and code as much as possible. There's a fundamental conflict between these features: on the one hand, sharing appears to imply no distinction between instantiations but on the other hand run-time types require it.

4.2 Object representation

Objects in the CLR's garbage-collected heap are represented by a vtable pointer followed by the object's contents (*e.g.* fields or array elements). The vtable's main role is virtual method dispatch: it contains a code pointer for each method that is defined or inherited by the object's class. But for simple class types, at least, where there is a one-to-one correspondence between vtables and classes, it can also be used to represent the object's *type*. When the vtable is used in this way we call it the type's *type handle*.

In an implementation of polymorphism based on full specialization, the notion of exact run-time type comes for free as different instantiations of the same parameterized type have different vtables. But now suppose that code is shared between different instantiations such as `List<string>` and `List<object>`. The vtables for the two instantiations will be identical, so we need some

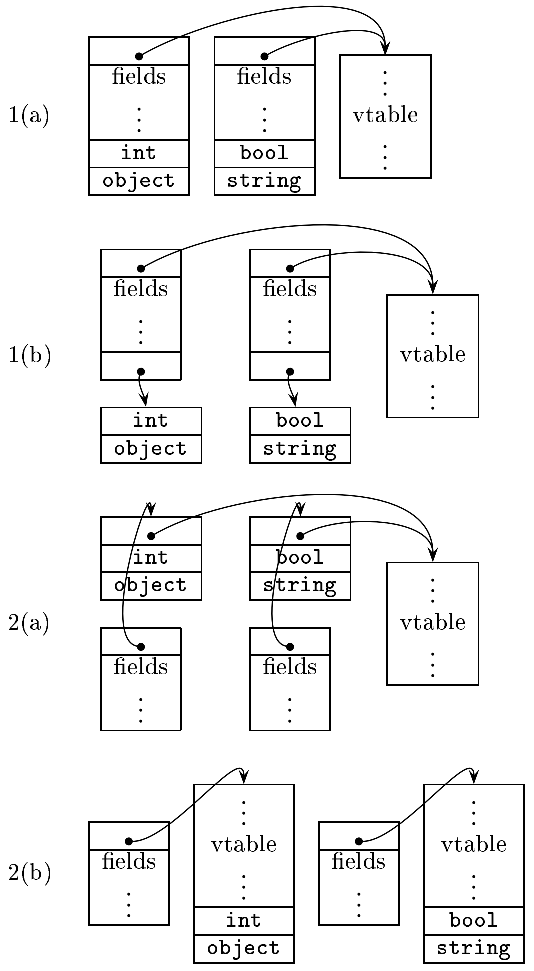

Figure 5: Alternative implementations of run-time types for objects of type `Dict<int,object>` and `Dict<bool,string>`

way of representing the instantiation at run-time. There are a number of possible techniques:

1. Store the instantiation in the object itself. Either
   (a) inline; or
   (b) via an indirection to a hash-consed instantiation.
2. Replace the vtable pointer by a pointer to a combined vtable-and-instantiation structure. Either
   (a) share the original vtable via an indirection; or
   (b) duplicate it per instantiation.

Figure 5 visualizes the alternatives, and the space and time implications of each design choice are presented in Figure 6. The times presented are the number of indirections from an object required to access the vtable and instantiation.

In our implementation we chose technique 2(b) because

- Polymorphic objects are often small (think of list cells) and even one extra word per object is too great a cost.
- We expect the number of instantiations to be small compared to the number of objects, so duplicating the vtable should not be a significant hit.

Technique	Space (words)		Time (indirections)	
	per-object	per-inst	vtable	inst
1(a)	n	0	1	0
1(b)	1	n	1	1
2(a)	0	$n+1$	2	2
2(b)	0	$n+s$	1	2

Figure 6: Time/space trade-offs for run-time types (n = number of type parameters and s = size of vtable)

```
class C { virtual void m(int) { ... } }
class D<T> : C {
  void p(Object x) { ... (Set<T[]>) x ... }
  override void m(int) { ... new List<T> ... }
}
class E<U> : D<Tree<U>> {
  // m inherited
  void q() { ... y is U[] ... }
}
```

Figure 7: Run-time types example

- Run-time types are typically not accessed very frequently, so paying the cost of an indirection or two is not a problem.
- Virtual method calls are extremely frequent in object-oriented code and in typical translations of higher-order code and we don't want to pay the cost of an extra indirection each time. Also, technique 2(a) is fundamentally incompatible with the current virtual method dispatch mechanism.

It should be emphasised that other implementations might make another choice – there's nothing in our design that forces it.

4.3 Accessing class type parameters at run-time

We now consider how runtime type information is accessed within polymorphic code in the circumstances when IL instructions demand it. For example, this occurs at any `newobj` instruction that involves constructing an object with a type that includes a type parameter `T`, or a `castclass` instruction that attempts to check if an object has type `List<T>` for an in-scope type parameter `T`. Consider the `Push` method from the stack class in Figure 1. It makes use of its type parameter `T` in an operation (array creation) that must construct an exact run-time type. In a fully-specialized implementation, the type `T` would have been instantiated at JIT-compile time, but when code is shared the instantiation for `T` is not known until run-time.

One natural solution is to pass type instantiations at run-time to all methods within polymorphic code. But apart from the possible costs of passing extra parameters, there is a problem: subclassing allows type parameters to be "revealed" when virtual methods are called. Consider the class hierarchy shown in Figure 7. The calling conventions for `C.m` and `D.m` must agree, as an object that is statically known to be of type `C` might dynamically be some instantiation of `D`.

But now observe that inside method `D.m` we have access to an instance of class `D<T>` – namely `this` – which has exact type information available at runtime and hence the precise instantiation of `T`. Class type parameters can then always be accessed via the `this` pointer for the current method.

Method `m` might be invoked on an object whose type is a subclass of an instantiation of `D` – such as `E<string>` which is a subclass of `D<Tree<string>>` – in which it is inherited. Therefore we must also include the type parameters of superclasses in the instantiation information recorded in the duplicated vtable (solution 2(b) above).

4.4 Instantiating open type expressions at run-time

Given we can access class type parameters, we must now consider when and where to compute type handles for type expressions that involve these parameters. These handles are required by instructions such as `newobj` and `castclass` mentioned above. For example, the code for method `D.m` in Figure 7 must construct a run-time representation for the type `List<T>` in order to create an object of that type.

For monomorphic types, or when code is fully-specialized, the answer is simple: the JIT compiler just computes the relevant type handle and inserts it into the code stream. A `newobj` operation then involves allocation and initialization, as normal.

However, for open type expressions (those containing type variables) within shared code, we have a problem. For example, if we use vtable pointers to represent type handles (see §4.2) then the implementation must perform a look-up into a global table to see if we've already created the appropriate vtable; if the look-up fails, then the vtable and other structures must be set up. Whatever representation of run-time types is used, the required operation is potentially very expensive, involving at least a hash-table look-up. This is not acceptable for programs that allocate frequently within polymorphic code.

Luckily, however, it is possible to arrange things so that the computation of type handles is entirely avoided in "normal" execution. As we will see later, we can achieve an allocation slowdown of only 10–20%, which we consider acceptable.

4.4.1 Pre-computing dictionaries of type handles

We make the following observation: given the IL for a method within a parameterized class such as `D`, all the sites where type handles will be needed can be determined statically, and furthermore all the type expressions are fully known statically with respect to the type parameters in scope. In class `D` there are two such open type expressions: `Set<T[]>` and `List<T>`. Given this information we can *pre-compute* the type handles corresponding to a particular instantiation of the open types when we first make a new instantiation of the class. This avoids the need to perform look-ups repeatedly at run-time. For example, when building the instantiation `D<string>`, we compute the type handles for `Set<string[]>` and `List<string>`. These type handles are stored in the vtable that acts as the unique runtime type for `D<string>`. That is, for each open type expression, a slot is reserved in a type handle *dictionary* stored in the vtable associated with a particular instantiation.

As with type parameters, these slots must be *inherited* by subclasses, as methods that access them might be inherited. For example, suppose that `m` is invoked on an object of type `E<int>`. The method code will expect to find a slot corresponding to the open type `List<T>`, in this case storing a type handle for `List<Tree<int>>`. So for the example class `E`, the information that is stored per instantiation in each the vtable has the layout shown in Figure 8.

4.4.2 Lazy dictionary creation

In classes with many methods there might be many entries in the dictionary, some never accessed because some methods are never invoked. As it is expensive to load new types, in practice we fill the dictionary lazily instead of at instantiation-time. The code sequence for determining the type for `List<T>` in method `D.m` is then

Class	Slot no.	Type parameter or open type
`D`	0	`T = Tree<U>`
`D`	1	`Set<T[]> = Set<Tree<U>[]>`
`D`	2	`List<T> = List<Tree<U>>`
`E`	3	`U`
`E`	4	`U[]`

Figure 8: Dictionary layout for example in Figure 7

```
vtptr = thisptr->vt;  // extract the vtable ptr
rtt = vtptr->dict[2]; // slot no. 2 in dictionary
if (rtt != NULL) goto Done; // it's been filled in
rtt = rtt_helper(...); // look it up the slow way...
vtptr->dict[2] = rtt;  // and update the dictionary
Done:
```

This is just a couple of indirections and a null-test, so hardly impacts the cost of allocation at all (see §5).

Moreover, laziness is crucial in ensuring termination of the loader on examples such as

```
class C<T> { void m() { ... new C<C<T>>() ... } }
```

which are a form of polymorphic recursion. In fact, it is even necessary to introduce some laziness into determining a type handle for the superclass in cases such as

```
class C<T> : D<C<C<T>>> { ... }
```

which in turn means that *inherited* dictionary entries cannot simply be copied into a new instantiation when it is loaded.

4.4.3 Determining dictionary layout

There is some choice about when the location and specification of the open type expressions is determined:

1. At source compile-time: compilers targeting IL must emit declarations for all open types used within a parameterized class definition.
2. At load-time: when first loading a parameterized class the CLR analyses the IL of its methods.
3. At JIT compile-time: the open types are recorded when compiling IL to native code.

We regard (1) as an excessive burden on source-language compilers and an implementation detail that should not be exposed in the IL, (2) as too expensive in the case when there are many methods most of which never get compiled, and therefore chose (3) for our implementation. The only drawback is that discovery of open type expressions is now incremental, and so the dictionary layout must grow as new methods are compiled.

4.5 Polymorphic methods

So far we have considered the twin problems of (a) accessing class type parameters, and (b) constructing type handles from open type expressions efficiently at run-time. We now consider the same questions for method type parameters.

As with parameterized types, full specialization is quite straightforward: for each polymorphic method the JIT-compiler maintains a table of instantiations it has seen already, and uses this when compiling a method invocation. When the method is first called at an instantiation, code is generated specific to the instantiation.

Once again for shared code things are more difficult. One might think that we often have enough exact type information in the parameters to the methods themselves (for example, using `arr` in the `Slice` method from Figure 3), but this is not true in all cases, and, anyway, reference values include `null` which lacks any type information at all.

So instead we solve both problems by passing dictionaries as parameters. The following example illustrates how this works.

```
class C {
  static void m<T>(T x) {
    ... new Set<T> ... new T[] ...
    ... p<List<T>>(...); ...
    }
  static void p<S>(S x) { ... new Vector<T> ... }
  static void Main() { ... m<int>(...) ... }
}
```

When `m` is invoked at type `int`, a dictionary of type handles containing `int`, `Set<int>` and `int[]` is passed as an extra parameter to `m`. The first handle is to be used for whenever `T` itself is required, and the latter two handles are ready to be used in the `new` operations of the method body.

Note, however, that more is required: `m` in turn invokes a polymorphic method `p`, which needs a dictionary of its own. We appear at first to have a major problem: the "top-level" invocation of `m<int>` must construct dictionaries for the complete transitive closure of all polymorphic methods that could ever be called by `m`. However, laziness again comes to the rescue: we can simply leave an empty slot in the dictionary for `m<int>`, filled in dynamically the first time that `p<List<int>>` is invoked. And, again, laziness is crucial to supporting polymorphic recursion in examples such:

```
static void f<T>(T x) { ... f<List<T>>(...) ... }
```

Finally we note that polymorphic *virtual* methods are altogether more challenging. If the code for a polymorphic virtual method was shared between all instantiations, as it is in non-type-exact systems such as GJ, then virtual method invocations could be made in the usual way through a single slot in the vtable. However, because the caller does not know statically what method code will be called, it cannot generate an appropriate dictionary. Moreover, in a system like ours that generates different code for different type instantiations, there is no single code pointer suitable for placement in a vtable. Instead, some kind of run-time type application appears to be necessary, and that could be very expensive. We will discuss efficient solutions to both of these problems in a future paper.

4.6 Verification

As with the existing CLR, our extensions have a well-defined semantics independent of static type checking, but a subset can be type-checked automatically by the verifier that forms part of the run-time. The parametric polymorphism of languages such as Core ML and Generic C# can be compiled into this verifiable subset; more expressive or unsafe forms of polymorphism, such as the covariance of type constructors in Eiffel, might be compiled down to the unverifiable subset.

The static typing rules for the extension capture the parametric flavour of polymorphism so that a polymorphic definition need only be checked once. This contrasts with C++ templates and extensions to Java based on specialization in the class loader [2] where every distinct instantiation must be checked separately.

Element type	Time (seconds)		
	Object	Poly	Mono
`object`	2.9	2.9	2.9
`string`	3.5	2.9	3.1
`int`	8.5	1.8	2.0
`double`	10.4	2.0	2.0
`Point`	10.5	4.3	4.3

Figure 9: Comparing Stacks

Type	Time (seconds)		
	Specialized	Runtime look-up	Lazy Dictionary
`List<T>`	4.2	288	4.9
`Dict<T[],List<T>>`	4.2	447	4.9

Figure 10: Costing run-time type creation

5 Performance

An important goal of our extension to the CLR was that there should be no performance bar to using polymorphic code. In particular, polymorphic code should show a marked improvement over the `Object`-based generic design pattern, and if possible should be as fast as hand-specialized code, another design pattern that is common in the Base Class Library of .NET. Furthermore, the presence of run-time types should not significantly impact code that makes no use of them, such as that generated by compilers for ML and Haskell.

Figure 9 compares three implementations of `Stack`: the C# and Generic C# programs of Figure 1 (Object and Poly) and hand-specialized variants for each element type (Mono). The structure of the test code is the following:

```
$S := new\ stack$
$c := constant$
$for\ m \in 1 \ldots 10000\ do$
  $S.push(c)\ m\ times$
  $S.pop()\ m\ times$
```

As can be seen, polymorphism provides a significant speedup over the crude use of `Object`, especially when the elements are of value type and hence must be boxed on every `Push` and unboxed on every `Pop`. Moreover, the polymorphic code is as efficient as the hand-specialized versions.

Next we attempt to measure the impact of code sharing on the performance of operations that instantiate open type expressions at run-time. Figure 10 presents the results of a micro-benchmark: a loop containing a single `newobj` applied to a type containing type parameters from the enclosing class. We compare the case when the code is fully specialized, when it is shared but performs repeated runtime type lookups, and when it is shared and makes use of the lazy dictionary creation technique described in §4.4. The run-time lookups cause a huge slowdown whereas the dictionary technique has only minor impact, as we hoped.

6 Related work

Parametric polymorphism and its implementation goes back a long way, to the first implementations of ML and Clu. Implementations that involve both code sharing and exact runtime types are rare: one example is Connor's polymorphism for Napier88, where exact types are required to support typesafe persistence [7]. Connor rejects the use of dynamic compilation as too slow – the widespread acceptance of JIT compilation now makes this possible. Much closer to our work is the extension to Java described by Viroli and Natali [16]. They live with the existing JVM but tackle the combination of code-sharing and exact run-time types by using reflection to manage their own type descriptors and dictionaries of instantiated open types, which they call "friend types". Such friend types are constructed when a new instantiation is created at load-time; the problems of unbounded instantiation discussed in Section 4.4 are avoided by identifying a necessary and sufficient condition that is used to reject such unruly programs. Our use of laziness is superior, we believe, as it avoids this restriction (polymorphic recursion can be a useful programming technique) and at the same time reduces the number of classes that are loaded. In a companion technical report [17] the authors discuss the implementation of polymorphic methods using a technique similar to the dictionary passing of Section 4.5. The observation that the pre-computation of dictionaries of types can be used to avoid run-time type construction has also been made by Minamide in the context of tagless garbage collection for polymorphic languages [12].

Other proposals for polymorphism in Java have certainly helped to inspire our work. However, the design and implementation of these systems differ substantially from our own, primarily because of the pragmatic difficulty of changing the design of the JVM. Agesen, Freund and Mitchell's [2] uses full specialization by a modified JVM class loader, but implements no code sharing. The PolyJ team [13] made trial modifications to a JVM, though have not published details on this. GJ [4] is based on type-erasure to a uniform representation, and as a result the expressiveness of the system when revealed in the source language is somewhat limited: instantiations are not permitted at non-reference types, and the runtime types of objects are "non-exact" when accessed via reflection, casting or viewed in a debugger. Furthermore, the lack of runtime types means natural operations such as `new T[]` for a type parameter `T` are not allowed, as Java arrays must have full runtime type information attached. NextGen [6] passes runtime types via a rather complex translation scheme, but does not admit instantiations at non-reference types. Pizza [14] supports primitive type instantiations but implements this by boxing values on the heap so incurring significant overhead.

7 Conclusion

This paper has described the design and implementation of support for parametric polymorphism in the CLR. The system we have chosen is very expressive, and this means that it should provide a suitable basis for language inter-operability over polymorphic code. Previously, the code generated for one polymorphic language could not be understood by another, even when there was no real reason for this to be the case. For example, it is now easy to add the "consumption" of polymorphic code (*i.e.* using instantiated types and calling polymorphic methods) to a CLR implementation of a language such as Visual Basic.

Potential avenues for future investigation include the following:

- Many systems of polymorphism permit type parameters to be "constrained" in some way. F-bounded polymorphism [5] is simple to implement given the primitives we have described in this paper, and, if dynamic type checking is used at call-sites, does not even require further verification type-checking rules. Operationally, we believe that many other constraint mechanisms can be implemented by utilizing the essence of the dictionary-passing scheme described in §4, *i.e.* by lazily creating dictionaries that record the essential information that records how a type satisfies a constraint.

- Our polymorphic IL does not support the full set of operations possible for analysing instantiated types at runtime: for example, given an object known to be of type `List<`τ`>` for some unknown τ, the type τ cannot be determined except via the CLR Reflection library. Instructions could be added to permit this, though this might require further code generation.

- Some systems of polymorphism include variance in type parameters, some safely, and some unsafely (*e.g.* Eiffel). Adding type-unsafe (or runtime type-checked) variance is clearly not a step to be taken lightly, and no source languages currently require type-safe variance.

- The absence of higher-kinded type parameters makes compiling the module system of SML and the higher-kinded type abstraction of Haskell difficult. We plan on experimenting with their addition.

- The presence of runtime types for all objects of constructed type is objectionable for languages that do not require them, even with careful avoidance of overheads. A refined type system that permits their omission in some cases may be of great value.

It would also be desirable to formalize the type system of polymorphic IL, for example by extending Baby IL [8].

With regard to implementation, the technique of §4.4 is absolutely crucial to ensure the efficiency of polymorphic code in the presence of runtime types. In effect, the computations of the handles for the polymorphic types and call sites that occur within the scope of a type variable are lifted to the point where instantiation occurs. Making this computation lazy is essential to ensure the efficient operation on a virtual machine.

In the future we would like to investigate a wider range of implementation techniques, and in particular obtain performance measurements for realistic polymorphic programs. The tradeoffs in practice between specialization and code sharing are only beginning to be properly understood (see [15, 2, 3] for some preliminary results). We have deliberately chosen a design and implementation strategy that allows flexibility on this point.

Acknowledgements

We would like to thank the anonymous referees for their constructive remarks. Also we are grateful to Jim Miller, Vance Morrison and other members of the CLR development team, Anders Hejlsberg, Peter Hallam and Peter Golde from the C# team, and our colleagues Nick Benton, Simon Peyton Jones and others from the Programming Principles and Tools group in Cambridge.

References

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

[2] O. Agesen, S. Freund, and J. C. Mitchell. Adding parameterized types to Java. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, pages 215–230. ACM, 1997.

[3] 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.

[4] 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.

[5] Peter S. Canning, William R. Cook, Walter L. Hill, John C. Mitchell, and William Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, 1989.

[6] 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.

[7] R. C. H. Connor. *Types and Polymorphism in Persistent Programming Systems*. PhD thesis, University of St. Andrews, 1990.

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

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Annual ACM Symposium on Principles of Programming Languages*, January 1995.

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

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

[12] Y. Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.

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

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

[15] Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza – Translating Parameterised Types into Java. Technical Report CIS-97-016, University of South Australia, 1997.

[16] M. Viroli and A. Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, October 2000.

[17] M. Viroli and A. Natali. Parametric polymorphism in Java through the homogeneous translation LM: Gathering type descriptors at load-time. Technical Report DEIS-LIA-00-001, Università degli Studi di Bologna, April 2000.