MSR White Paper: Proposed Extensions to COM+ VOS (Draft)

Don Syme, Nick Benton, Simon Peyton-Jones, Cedric Fournet

1 Introduction

1.1 Getting Serious about Language Innovation

A primary objective of the COM+ Runtime is to deliver services and performance that are clearly technically superior to those provided by other potential backend runtime environments. By doing this COM+ will become the backend environment of choice for compiler writers and users on supported platforms. One implication of this is that the COM+ Runtime must provide the features required for the efficient execution and seamless interoperability of advanced programming languages such as Standard ML, Haskell, Cool and future versions of Java. COM+ will then support these programming languages with improved overall functionality and will lower the total costs for compiler writers and developers.

This document proposes a set of extensions to the COM+ Runtime which are required to provide quality support to advanced programming languages. The extensions are mostly related to the Virtual Object System (VOS), i.e. the type system of COM+. The aim of this document is to describe a concrete proposal for each of these features, taking into account the current state of the COM+ Runtime. It is easy to make half-baked proposals for these sorts of extensions – instead we have worked through the proposals in detail, in the context of the COM+ Runtime, and have provided lengthy examples, so the proposals can be evaluated on their merit.

The increased expressiveness provided by these features can lead to less code, faster execution, better libraries, better components, and support for advanced source languages. The guarantees provided by these features can also lead to the faster execution of programs, because, as usual, type correctness can allow the omission of some checks at runtime. The features come with negligible runtime cost and are nearly entirely backward compatible with the existing COM+ design.

The features considered in this document are:

- Compound Types (CT, Section 2)
- Parametric polymorphism (PP, Section 3)
- Covariant Return Types (CRT, Section 4)
- Generalized Delegates (i.e. Function Types, Thunks and Closures) (Section **Error! Reference source not found.**)
- Enhanced Parametric Polymorphism, including type variance and instantiation by unboxed types (EPP, Section 6, *TBD*)

Parametric polymorphism is widely regarded by developers and academics as the most important possible extension to the core of the Java language. For example, to quote from Steele and Cartwright [SC98]:

> *"The most serious impediment to writing substantial programs in the Java programming language is the lack of a genericity [i.e. PP] mechanism for abstracting classes and methods with respect to type."*

The primary reason why this feature has not yet been officially added to Java is the JVM severely constrains its delivery. For example, [SC98] documents the horrible contortions a compiler writer must go through in order to compile PP to the existing JVM.

Microsoft should seize the initiative on these issues, both in the design of the COM+ Runtime and Cool, and regardless of whether the current version of Java does or does not include support for PP and the other features described here. It is almost certain that future versions of Java will support these

features. Microsoft should be the first to deliver the basic services that are required to support such language features, whether from VB, Java, Cool, Standard ML, Haskell or Ada.

In addition, it would not be difficult to exploit these features within the Common Language Subset (CLS). This would greatly improve the COM+ library designs and the quality of each CLS compliant programming language, in the same way that the the C++ template mechanism led to the highly regarded C++ Standard Template Library.

To summarize, the features we propose will:

(a) Allow components to describe their contract of behavior in a more detailed and precise way. Important aspects of the contract, such as genericity, can then be described and checked statically.

(b) Permit more efficient execution for languages that support these features. This is because successful type checking in a stronger type system allows the omission of many runtime checks.

(c) Facilitate the delivery of an improved set of libraries (standard components). The features allow library components to deliver stronger guarantees and allow components to be reused more flexibly.

(d) Improve the CLS by allowing interoperability between those high level languages that that can compile similar features uniformly to the representations described here.

(e) Improve the quality of the source languages supported on Microsoft platforms, and put Microsoft at the forefront of delivering advanced, high productivity features of programming languages to developers on COM+ platforms.

1.2 Proposed Schedule

The table below breaks down the proposed mechanisms into individual design elements, and summarizes the relevance (languages supported) and estimated cost (in man months) of each feature. Shaded items are those we believe should be immediately scheduled for inclusion in COM+, the CLS and Cool, taking into account all factors such as cost, relevance, immediate and long term benefits. The sections of the COM+ implementation that would be effected are shown, as are the primary reasons why the feature is worth adding:

✓Components: For better components in the CLS, and to improve VB, VC++ and VJ.

✓Cool: To make Cool a better language

✓Languages: To support the type systems of other languages, such as those found in Project 7

✓Efficiency: For more efficient execution of other languages, such as those found in Project 7.

1.2.1 Compound Types

Feature	*VOS/CLS/Cool*		*Languages that Benefit*	*Benefits to Microsoft*
	Effects	*Work Weeks*		
CT-1. Represent Compound Types	Metadata Assembler Reflection	2	?VC++ ?Cool ?VB ?VJ	✓Components ✓Cool
CT-2. Verifier Checks	Metadata Verifier	4		
CT-3. Add to CLS & Cool	CLS Cool	12		

1.2.2 Covariant Return Types

Feature	***VOS***		*Languages that Benefit*	*Benefits to Microsoft*
	Effects	*Work Weeks*		
CRT-1. Add to COM+	Metadata Loader Assembler Verifier	4	?VC++ ?Cool ?VB ?VJ	✓Components ✓Cool
CRT-2. Add to CLS/Cool	CLS Cool	4		

1.2.3 Function Types, Closures and Thunks (= Generalized Delegates)

Feature	***VOS***		*Languages that Benefit*	*Benefits to Microsoft*
	Effects	*Work Weeks*		
GD-1. Add Function Types to COM+	Class Libs	2	✓SML ✓CaML ✓Haskell ✓Scheme ✓Mercury ?Java ?Cool ?VB ?VC++	✓Components ✓Cool ✓Languages ✓Efficiency
GD-2. Optimize Closure Template Descriptions	Metadata PreJIT Assembler Verifier	4		
GD-3. Align with Delegates	VOS PreJIT JIT GC Class Lib Cool VC++ VB	8		
GD-4. Optimized Closures	PreJIT	8		
GD-5. Optimized Thunks	PreJIT GC	8		

1.2.4 Parametric Polymorphism

Feature	***VOS***				***CLS & Cool***			*Languages that Benefit*	*Benefits to Microsoft*
	Effects	*Man Months*	*Now*	*Later*	*Man Months*	*Add!*	*Plan to add?*		
PP-1. Represent Parametric Types	Metadata Assembler Verifier Reflection	2	Yes		1	Yes		✓SML ✓Haskell ✓Mercury ?Cool ?Java ?VB ?VC++	✓Components ✓Cool ✓Languages ✓Efficiency
PP-2. Parametric Code	PreJIT Metadata Assembler Verifier GC	12	Maybe	Yes	12	Maybe	Yes		

PP-3. New PP Libraries	Class Lib	6	Maybe	Yes	6	Maybe	Yes		
PP-4. Bounded PP Types & Code	Metadata Verifier	3		Maybe	3		Maybe		
PP-5. Base Type Code-Expanding PP	PreJIT Verifier	6		Maybe	6		Maybe		
PP-6. Variant Types Parameters	Metadata Verifier Reflection	6		Maybe	6		Maybe to Cool		

1.3 Halfway is Not Good, but may have to do

While the authors would press for the COM+ Runtime to provide full support for the features described here, we understand that resource and schedule pressures will make it difficult to do so in COM+ 2.0. We strongly urge the COM+ and CLS designers not to take any design decisions that would prohibit the later adoption of the features described here.

It is possible Microsoft can provide partial support for some of these features "by agreement". The feature would then be implemented by tagging COM+ modules with metadata declarations that indicate their behavior with respect to an enhanced type system. As far as the COM+ Runtime is concerned, e.g. with respect to verifiability, the declared contract is actually weaker. However in order to access such components, compilers or programmers would have to insert appropriate runtime casts and typechecks.

This is similar to the current situation with the JVM, and results in inefficient execution, confusion, weaker behavioral guarantees and code bloat. Benefit (b) above would not then be realized, and the other benefits would be harder to achieve because the cost of supporting each feature is shifted to the compiler writer. Unless very clear rules are laid down for the compilation and use of such features, integration between languages would suffer dramatically. Furthermore, features such as runtime type querying and reflection would not be seamlessly supported, leading to a substantial loss of coherence.

However, even if the COM+ team does not adopt these proposals, some version of them, perhaps substantially modified, will live on via Project 7, implemented via this less satisfactory route.

2 Compound Types

We propose that the COM+ VOS type system should be modified to support "compound types" as described by Weck and Büchi (http://www.tucs.fi/publications/techreports/ TR182.ps.gz).[1] Compound types are also known as intersection types or interface sets.

2.1 Very Simple and Very Useful

A compound type is made up of several interface types and an (optional) class type. An object that is a member of a compound type must support all the contracts specified. They are most useful when an argument must support several interfaces, but we do not want to name the combination of these interfaces, e.g.

```
public static foo([IFoo, IGoo] x) {
   x.IFoo::Method1();
   x.IGoo::Method2();
}
```

Here "x" has the compound type [IFoo, IGoo] and must support both of these interfaces. The advantage is that we don't need to create a new named interface to represent the combination of IFoo and IGoo. This means that clients can call this method using an "x" that happens to support both IFoo and IGoo, without explicitly declaring that it also supports the combination of these interfaces.

Compound types are *not* just notational shorthand, because it becomes possible to pass arguments to a method such as "foo" that have not been explicitly declared to implement some interface that is a combination of IFoo and IGoo.

The advantages of compound types are:

1. They permit multiple interface inheritance without forcing users to explicitly name those interfaces that combine several other interfaces.

2. They greatly increase the reusability and compositionality of components that are described and accessed via interfaces. In COM+, this includes the vast array of components that will be accessed via the COM interop mechanism.

3. They are fully backward compatible with existing features of the COM+ type system. In particular, they require no modifications to the class vtable layouts or the dynamic dispatch mechanisms.

4. They interact elegantly with the parametric polymorphism described in Section 1, because they may be used to describe multiple bounds for a type parameter.

The essence of Weck/Büchi proposal is as follows:

- Compound types are anonymous (i.e. not named). We've seen this before with array types: you can simply use "`String[]`" rather than defining "`class MyStringArray = ...`". The order of interfaces is also irrelevant. [IFoo,IGoo] at one place in a program is considered identical to [IGoo,IFoo] in another.

- Values that conform to compound types must support the behavior of each type in the compound type. That is, an argument of type [IText,IContainer] must, at runtime, be an instance of a class that implements both IText and IContainer (and perhaps others).

[1] The compound types described here are not to be confused with structured types (records, arrays and structs), which are sometimes also called compound types.

- Compound types may be used wherever interface types are currently used, except: (a) in the "implements" declaration of a class; or (b) in the "extends" declaration of an interface, or (c) when identifying a method to call. You may cast to a compound type.

- Compound types do not change the runtime type of an object. They are only used to describe static types.

- Equivalence of compound types is by *structure* and not *name*. We've seen this before with array types: two type tokens that both describe String[] are considered identical if they have the same structure, even if their two type tokens are not identical.

- Subtyping extends to compound types in the natural way: a compound type is a supertype of any type that supports all of the listed behaviors, and is a subtype of any type that supports fewer behaviors.

- The least upper bound of two compound types is simply a new compound type that is made from the least upper bound of the classes in each type (considered to be Object if there is no class) and the intersection of the sets of supported interfaces.

2.2 An Example

To quote from the Weck/Büchi proposal:

> *"Frequently, classes need to conform to more than one contract. For instance, Microsoft's OLE defines ActiveX controls via a bundle of contracts to be implemented. Java supports multiple subtyping to this end. Similarly, one may want to declare variables or method parameters of a type comprising several contracts. This is not supported to the same degree by Java. On a first glance it may seem no problem, because one would only have to declare the right subtype. We will demonstrate, however, that this may not be possible with independently developed software components."*
>
> *"The following example describes a situation that cannot be properly handled by Java's type system. We assume two different and independent standards, which have come into existence entirely unrelated. One of them defines an interface* ***Text****, describing operations, such as to insert or delete characters. We also assume a function computing where in a view a specific character would be placed. The second standard defines a compound document framework, like OLE, including an interface Container to be implemented by all objects that may act as a compound document container. The latter must support insertion and removal of document parts.*

```
/* as part of a text framework */
public interface Text {
  void insert (char ch, int textPos);
...
  java.awt.Point displayPoint (int textPos);
}

/* as part of a compound document framework */
public interface Container {
   void insertPart (DocPart part, java.awt.Point xyPos);
  ...
}
```

> *Both standards form individually useful frameworks. Vendors can build components for either of them. The problem comes with the wish to create components that build on both standards simultaneously.*
>
> ...

```
/* Vendor A's component */
```

```
public class TextContainerA implements Text, Container {
  ...
}

/* Vendor B's component */
public class ContainerTextB implements Container, Text {
  ...
}
```

*[The above] shows portions of two sample classes, **TextContainerA** and **ContainerTextB**, provided by two different vendors. These classes exhibit a little nuisance. To insert a document part one has to pass the graphical coordinates because the container interface must be used. One may prefer to give a text position and have the part inserted after the corresponding character. For this purpose, a generic service can be implemented that maps the text position to a display position and then inserts the document part there. We assume that a Vendor C wants to offer this service within a class **LibraryServices**.*

```
/* Vendor B's component */
public class LibraryServices {
  public static void insertDocPart (DocPart part,
                                    ?? into,
                                    int textPos)
    { ... }
 /* the question marks stand for a type saying that interfaces Text and Container
    must be implemented */
}
...
```

*The obvious solution is to create a combined interface **TextContainer**, which extends both **Text** and **Container**, and does not add or hide anything...However, neither instances of **TextContainerA** or **ContainerTextB** are compatible with the library service, as they are not declared to implement the combined interfaces.*

The proposal goes on to describe how compound types are the appropriate solution to this problem.

2.3 Compound Types are needed anyway

Compound types are needed internally in the COM+ verifier in order to remove a source of incompleteness. In particular, the verifier fails when finding the least upper bound of two interface types such as IKoo and IMoo, declared as

```
interface IKoo extends IFoo,IGoo,IHoo { ... }
interface IMoo extends IFoo,IHoo { ... }
```

Morally, the least upper bound of these two types should be [IFoo,IHoo], i.e. the least upper bound of two otherwise unrelated interfaces is the intersection of all the interfaces included by those interface types. At the moment the COM+ verifier fails. For example, the following code may fail to verify:

```
public foo(IKoo x, IMoo y, boolean b) {
       ldarg #b
       bge path2
path1: ldarg #x
       br join
path2: ldarg #y
       br join
join:  dup
       call IFoo::Method1()
       call IHoo::Method1()
       ret
}
```

The verifier must take the least upper bound of the two possible types for the argument on the stack at location "join". These types are IKoo and IMoo. Currently this operation will fail. Even if Object is chosen as the least upper bound, verification will then fail for the first **call** instruction, since Object does not support IFoo::Method1().

Similarly, the least upper bound of the following two types

```
class Koo implements IFoo,IGoo,IHoo { ... }
interface IMoo extends IFoo,IHoo { ... }
```

should be [IFoo,IHoo]. The same applies to:

```
class Koo implements IFoo,IGoo,IHoo { ... }
class Moo implements IFoo,IHoo { ... }
```

2.4 Implementation Path A – Full Support

Full support for compound types is not difficult:

- Represent them in COM+ metadata in a similar manner to array types.
- Modify two procedures in the verifier: the one that checks subtyping, and the one that finds the least upper bound of two subtypes. The basic verifier algorithm need not change.
- Modify the reflection library in the obvious way.
- Modify the runtime typechecks performed by **cast** and **instanceof** when the argument token describes a compound type.

2.5 Implementation Path B – Support by Agreement

Providing full support for compound types is not difficult. However, it is possible to support compound types in a source language by compiling them down to weaker type signatures and a corresponding set runtime casts and checks. This is non-optimal because different languages must agree to use the same compilation mechanisms, metadata declarations and must insert appropriate casts, thus reducing the utility of compound types. In addition, no COM+ libraries will be able to use compound types, and the incentive to include compound types in Microsoft supported languages such as Cool will be reduced.

Weck and Büchi describe how compound types may be supported while using the existing JVM. Basically, wherever a compound type [IFoo,IGoo,IMoo] is used in the Java source language, the actual type in the metadata of the JVM code is some common supertype of all of these, e.g. Object.[2] Metadata declarations would still be required in IL executables to indicate their type signatures with compound types included. The COM+ Runtime would ignore these.

Whenever a value of such a type is actually accessed, a cast must first be performed to the appropriate bound at which the type is being accessed, e.g. if calling a method IGoo::goo() on such an object, we first insert the machine code "**cast** IGoo". The bytecode is then typesafe, and none of the casts will fail at runtime if the program has been compiled correctly and is passed data of the correct form.

However, if other components "cheat", and send an object that does not support the interfaces in the compound type, then spurious **InvalidCast** exceptions will be raised, and thus the behavior that the programmer expects by looking at the Java source code may not actually occur at runtime. The resulting is system is still typesafe, but unexpected casting errors may occur.

[2] Weck and Büchi actually propose using one of the listed types, even if it is not a common supertype.

3 Parametric Polymorphism

3.1 Introduction

Parametric Polymorphism (PP) allows classes and methods to be abstracted with respect to types. Most readers will be familiar with C++ templates, Ada generics, ML-style polymorphism or one of the proposals for genericity in Java (e.g. GJ, see http://www.cs.bell-labs.com/~wadler/pizza/gj). The PP mechanism we propose here is designed to deliver runtime and component- level support for these mechanisms.

PP is valuable because it promotes component reuse. It does this by greatly increasing the accuracy with which generic components can describe the contract they implement. PP allows components to express constraints that otherwise can only be enforced by runtime type checks. PP encourages the reuse of components that are truly generic, and when used systematically can lead to lower memory usage.

Components that describe their genericity statically allow client components to perform more static checks when they use the component. This means many more bugs can be caught at compile time, which greatly reduces the cost of software development. PP also reduces the number of classes that must be declared for naturally generic constructs such as delegates and closures, again leading to lower memory usage.

As it stands, genericity can only be achieved in COM+ and Java via the type Object. Runtime casts are then needed to and from appropriate types. This leads to both a loss of static information in the signatures of components as well as a performance hit. PP can lead to better runtime performance, because fewer casts are required when manipulating generic structures.

To quote from Guy Steele, one of the Java designers:

> *In our experience, the most serious impediment to writing substantial programs in Java is the lack of a mechanism for abstracting classes and methods with respect to type. Type parameterization can be simulated in Java using the universal reference type Object in place of type parameters and explicitly casting values of type Object to their intended types. But this coding idiom is clumsy and error prone because the required casts are tedious to write and largely defeat the error-detection properties of Java's static typing discipline. [CS98].*

The fact that this is Java is neither here nor there: the same problems have arisen in every high level programming language.

It would be regrettable for COM+ to make the same mistakes as the JVM and head down a path that cannot easily support PP. At the very least,we encourage the COM+ designers to ensure that their existing design choices will be compatible with the design proposed here.

3.1.1 PP, Source Languages and the CLS

This document does not constrain a source language to a particular kind of PP. Rather, it describes a very general form of PP that should be suitable for compiling the genericity mechanisms of most high level languages, perhaps in conjunction with some additional runtime typechecks for the more exotic varieties of parametricity that exist.

Similarly, this document does not specify the kind of PP that might be chosen for the CLS. The CLS may use a restricted kind of PP, or none at all. Thus, as for the existing VOS (Virtual Object System), this PP proposal is the raw material out of which a PP mechanism for the CLS could be built.

3.1.2 Overview

The PP mechanism proposed here is a modified version of that proposed for Java (as a source language) by Wadler et al. (http://www.cs.bell-labs.com/~wadler/pizza/gj/). The key features are summarized below. We will explain the technical details in later sections: this summary is included for those familiar with existing parametricity mechanisms in class based languages:

- The level of expressiveness is somewhere between C++ templates (very expressive, but they require code expansion) and ML-style polymorphism (limited expressiveness, no code expansion).

- Type parameters may be specified for classes, interfaces, value classes, and methods of all kinds.

- Type variables may normally only be instantiated by boxed types (i.e. Object or any of its subtypes). This is because different instantiations must have uniform representations. Unboxed type variables including base types and value classes are allowed, but may only be used in very limited circumstances, because different instantiations may have different representations.

- Type variables may be bounded, indicating a minimal contract that the variable must support. Bounds on types may be mutually recursive across a set of variables being introduced. The default bound on a type variable is Object.

- Instantiations of type variables are not erased at runtime. That is, the runtime type information available for an object includes the instantiations given to type parameters. This means that objects belonging todifferent two instantiations of a polymorphic type (e.g. "List<Object>" and "List<String>") can be distinguished at runtime without looking at the fields of the objects. See Section 3.5.5.

- Casts and serialization of parametric data are fully supported.

- Static fields may not be polymorphic. Static methods do not by default acquire their enclosing class's type parameters, though the parameters may be re-specified for each static method. See Section 3.5.

- Type parameters are non-variant. Section 6.1 proposes a possible variance mechanism.

- No data is required in an executable file for each ground instantiation of a parametric type. This is the primary advantage of this mechanism over PP proposals for Java that try to utilize the existing JVM.

- The instantiations of type variables do not effect dynamic dispatch. In other words, the eventual target of a dispatch does not depend on the actual instantiations of type variables.

- The COM+ Runtime need only JIT each method body once. However extra versions of code may be generated, at the discretion of the runtime, in order to optimize particular representations of particular instantiations.

3.2 Parametric Types

The first step to adding parametricity is simply to add the ability to represent types (classes, interfaces and value types) that are *parameterized* by other types. For example, List<String> is an instantiation of the generic type List<*T*>. We call "List" a *parametric* (or *generic*) *type*, or sometimes a *parametric type constructor*.

Several parametric type constructors already exist in COM+:

- Arrays: "*T*[]", "*T*[,]", ... Multi-dimensional arrays can be thought of as a single parametric type constructor with an additional integer parameter specifying the rank of the array.

- Pointers: "**T*"
- Byrefs: "*&T*"
- Boxing of value classes: "Box(*T*)"
- Function Pointers: "*T* &(*Arg1*)", "*T* &(*Arg1*, ..., *ArgN*)", ... Function pointers are best thought of as a single parametric type constructor with a variable number of type parameters.

In addition, both Compound Types (Section 2) and Function Types (Section 5) are most naturally represented in the same way as function pointers, i.e. as *n*-ary parametric type constructors.

Because so many parametric types already occur in COM+, it would be worth rationalizing the mechanisms used to implement these types, even if the rest of this PP proposal is not adopted.

This document doesn't describe how parametric types and their instantiations should be represented in the metadata of a COM+ PE file. However, it is clear that it a straightforward generalization and simplification of the current mechanisms used to specify the above types is possible, and is likely to be the best course to follow.

How we *represent* such types in a PE file is quite a different issue from how we describe *implementations* of these types, which is different again from how we *represent the implementations at runtime*. The following section describes several examples of how we describe implementations of a number of these types.

3.3 Examples of Implementing Parametric Types

3.3.1 Collection Classes

No classes illustrate PP better than standard collection classes. The following examples describe portions of the types such as Enumerator, Collection and List taken from Microsoft/Runtime/Collections, somewhat simplified for the sake of the presentation, and re-interpreted using PP.

We use a mixture of C++, Java and COM+ notation. We write code sequences as Java pseudo-code, assuming that the PP mechanism is exposed in Java as in GJ. Because we are dealing with the COM+ Runtime, the Java code itself does not really matter – the COM+ IL assembly code is more important. This is included where relevant, in a modified version of the syntax of the IL assembler.

3.3.1.1 Enumerators

We begin with an abstract parametric enumeration class:

```
abstract class Enumerator<T>
{
    public abstract boolean GetNext();
    public abstract T GetObject();
    public abstract void Remove();
}
```

The first line introduces ("binds") the type parameter *T*, which can be used as a type inside the body of the class Enumerator. There are no surprises here to those familiar with existing parametricity mechanisms, as the class is polymorphic in a very simple way.

When *T* = Object, i.e. the type Enumerator<Object>, then we get something very close to the current COM+ Enumerator class, which looks like this:

```
// The current COM+ enumerator class looks like this:
abstract class Enumerator
```

```
{
    public abstract boolean GetNext();
    public abstract Object GetObject();
    public abstract void Remove();
}
```

When *T* is something more specialized than Object, e.g. String, then the verifier ensures that GetObject() will indeed always return a String. The more precise type signature means that more errors can be caught at compile time. It also means we can omit the runtime typecheck associated with casting from Object to String each time GetObject() is called.

It is likely that we will be able to use the trick of erasing type parameters and replacing them by Object in order to achieve seamless interoperability for languages do not support PP. Then COM+ would consider the existing COM+ type Enumerator to be precisely the same as Enumerator<Object>.

3.3.1.2 Calling an Enumerator

The following method shows how we may use the parametric enumerator class for some particular instance of *T*, in this case *T* = String.

```
//add up the lengths of all the strings in the enumeration
global static int MyTotalLengths(Enumerator<String> myEnum)
  Java:
    {
      int n = 0;
      while (myEnum.GetNext()) {
          n+=myEnum.GetObject().GetLength();
      }
      return n;
    }

  IL:
    .local int n
    .local String s
          ldc 0
          stloc #n;
    loop: ldarg #myEnum
          call "Enumerator<String>::GetNext()"
          beq done
          ldarg #myEnum
          call "Enumerator<String>::GetObject()"
          call "String::GetLength()"
          ldloc #n
          add
          stloc #n
          br loop
    done: ldloc #n
          ret
```

When specifying an argument such as "myEnum", we must specify what kind of Enumerator it is. This means a value must be given for the type parameter *T*, and we call such a value an *instantiation*. In this case the instantiation is Enumerator<String>. We may *not* declare an argument to be of an uninstantiated type such as "Enumerator". The parallel with procedure calls is worth noting - we have *formal* and *actual* type parameters. In the above example the formal type parameter is *T* and the actual is String.

When accessing an instance method (or field) of a polymorphic type, the method reference in the IL code must specify completely which instantiation of the class is being accessed. This means the method reference specifies "Enumerator<String>::GetNext()" rather than "Enumerator::GetNext()". Thus method and field accesses are "fully resolved" at the level of IL code. Static members are discussed in Section 3.4.6. The rules for referencing members are described in Section 3.5, along with issues related to overloading and overriding. We discuss how instantiations might be represented in COM+ metadata in Section 3.5.5.

3.3.1.3 Implementing an Enumerator

The simplest implementation of an enumerator traverses a specified array:[3]

```
class ArrayEnumerator<T> extends Enumerator<T>
{
    private T[] items;
    private int where;
    public constructor ArrayEnumerator(T[] x)
       Java:
         { items = x; where = -1; }
       IL:
         ldarg #this
         call "Microsoft/Runtime/Object::init()"
         ldarg #this
         ldarg #x
         stfld "ArrayEnumerator<T>::items"
         ldarg #this
         ldc -1
         stfld "ArrayEnumerator<T>::where"

    public boolean GetNext()
       { return (++where >= items.length); }

    public T GetObject()
       { return items[where]; }

    public void Remove()
       { throw new NotSupportedException(
                    "This Enumerator can't remove items."); }
}
```

When extending a class we write "Enumerator<*T*>", since we are extending Enumerator for all *T*.

The IL assembly code is shown for the ArrayEnumerator constructor. Note that the specification of fields includes the class where the field occurs, and the exact type parameters for the instantiation of the class that we are accessing.[4]

3.3.1.4 Implementing one particular kind of Enumerator

We may also implement a parametric class at a particular instantiation. For example, the following (rather contrived) example is an enumerator for all vowels as strings:

```
class Vowels extends Enumerator<String>
{
    private static String[] data = ["A","E","I","O","U"];
    private int where;
    public constructor Vowels() { where = -1;}
    public boolean GetNext()
       { return (++where >= 5); }
    public String GetObject()
       { return data[where]; }
    public void Remove()
       { throw new NotSupportedException(
                    "This Enumerator can't remove items."); }
}
```

Of course better might be:

---

[3] This example also illustrates how type parameters can be used to specify Java array types, which has been problematic for PP mechanisms such as GJ. This is because these use "type erasure", i.e. they have no runtime type information for the type parameters. Java style array types necessarily require some type information at runtime, even to simply create them.

[4] We have not shown how method and field references are represented in COM+ metadata, which is primarily an implementation decision, discussed briefly in Section 3.5.1.

```
global final static String[] data = ["A","E","I","O","U"];
global static method Enumerator<String> Vowels()
     Java:
         { return new ArrayEnumerator<String>(data); }
     IL:
         ldsfld data
         newobj "ArrayEnumerator<String>::ArrayEnumerator(T[])"
         ret
}
```

This also demonstrates that ArrayEnumerator<String> is a subtype of Enumerator<String>, because the method may safely return the former in place of the latter. Subtyping in the presence of polymorphism is defined in Section A.1.6.3.

3.3.1.5 Collections and Lists

The next example is part of a parametric version of the abstract Collection class, which is used for objects that provide default enumeration behavior:

```
abstract class Collection<T>
{
    public abstract int GetSize();
    public abstract Enumerator<T> GetEnumerator();
    ...
}
```

The abstract class List then adds the ability to randomly access and insert into collections:

```
abstract class List<T> extends Collection<T>
{
    public abstract void Clear();
    public abstract T Get(int index);
    public abstract void Insert(T value, int index);
    public Enumerator<T> GetEnumerator()
      { return new ListEnumerator<T>(this); }

}
```

The implementation of the default iteration behavior is via the ListEnumerator class:

```
abstract class ListEnumerator<T> extends Enumerator<T>
{
    private List<T> items;
    private int where;
    public constructor ListEnumerator(List<T> x)
      { items = x; where = 0;}
    public boolean GetNext()
      { return (++where >= items.GetSize()); }
    public T GetObject()
      { return items.Get(where); }
    public void Remove()
      {  throw new NotSupportedException(
                    "This Enumerator can't remove items."); }
}
```

3.3.1.6 Lists represented by Arrays

We can implement List by pinning down the representation to be an expandable array of items. We omit most the details as the implementation is straightforward.

```
class ArrayList<T> extends List<T>
{
    private T[] items;
    private int size;
```

```
    public ArrayList(int capacity) { ... }
    public abstract void Clear() { ... }
    public abstract T Get(int index) { ... }
    public abstract void Insert(T value, int index) { ... }


}
```

This raises the question of the subtyping relationship between ArrayList<*T1*> and List<*T2*>. Certainly ArrayList<*T*> is a subtype of List<*T*> for all *T*. This is *non-variant* subtyping. We discuss an optional mechanism for *co-variant* and *contra-variant* subtyping in Section 6.1, but in the basic proposal we limit ourselves to non-variant subtyping.

3.3.2 Polymorphic Methods

Let us assume we want to write a static function to count the length of a list for any List<*T*>. Assuming we cannot change the code for the class List<*T*>, we add this to a class of static library functions called "MyLib".

```
class MyLib
{
   ...
   public static int Length<T>(List<T> x)
        {
          int n = 0;
          Enumerator<T> i = x.GetEnumerator();
          while (i.GetNext()) {
              n++;
          }
          return n;
        }
}
```

The method "Length" is polymorphic in its own right, and the class "MyLib" is *not* polymorphic. When a method is polymorphic we write its type variables immediately after its name. This is why we write "int Length<*T*>(...)" instead of "int Length(...)".

The scope of the type variables declare for a method includes the return type of the method. This means a method signature such as "*T* Head<*T*>(List<*T*>)" is allowed:

```
class MyLib
{
   ...
   public static T Head<T>(List<T> x)
        {
          Enumerator<T> i = x.GetEnumerator();
          i.GetNext();
          return i.GetObject();
        }
}
```

Static methods are not polymorphic by default. That is, class type variables may not be used in static methods and static fields (this is discussed in Section 3.5) unless the polymorphism is explicitly declared. When static methods are polymorphic, they must list their type variables after the method name just like any other polymorphic method. Thus, even if the static member Length were defined in the class List<*T*>, we would still declare it to be explicitly polymorphic:

```
class List<T>
{
   ...
   public static int Length<T>(List<T> x) { ... }
   ...
}
```

To put it another way, *static members are not included under the scope of class type variables*. The underlining below indicates the scope of class type variables:

```
class C<T1, ..., Tn> extends ... implements ...
{
    // T1, ..., Tn may be used in any of the following:
    Instance Fields
    Instance Methods
    Constructors
    Properties
    Events

    // T1, ..., Tn may not be used in the following.
    // If a static method is polymorphic it can
    // declare T1, ..., Tn to be type parameters of the
    // method.
    Static Fields
    Static Methods
}
```

This is because static methods are shared by all instances, and many static methods are not naturally polymorphic in the same way as their class.

Non-static methods may also be polymorphic, though these are less common. Static fields may never be polymorphic.

3.3.3 Comparison and Sorting

The next example shows a parametric interface for comparing values. An outline of a class that implements this interface for *T* = String is also shown.

```
interface IComparer<T> {
    public int Compare(T x, T y);
}

class Lexical implements IComparer<String> {
    public int Compare(String x, String y)
       { ... lexical comparison of string ... }
}
```

In contrast, the existing COM+ IComparer class has signature:

```
//The existing COM+ IComparer interface is:
interface IComparer {
    public int Compare(Object x, Object y);
}
```

and the equivalent of "Lexical" would have to cast from Object to String before doing the real work of comparison. Again parametricity allows us to omit runtime typechecks and catch more errors statically.

3.3.3.1 A Sort Method

Parametric comparison can be used to add a Sort method to the List class.

```
abstract class List<T> extends Collection<T>
{
    ...

    public int Sort(IComparer<T> comparer) {
      //sort the list, comparing using "comparer.Compare(a,b)"
      ...
    }
}
```

3.3.4 Printable Lists via Bounded Parameters

Let us assume the interface IPrintable represents a contract to print a representation of the object to standard output.

```
interface IPrintable
{
        public void Print();
}
```

We can now define extra functionality on Lists to print an entire list. The appropriate Clone() method will be called on each and every element of the list. This then also becomes the default cloning behavior of our new list class.

```
abstract class PrintableList<T : IPrintable>
                            extends List<T>
                            implements IPrintable
{
    ...
    public void Print()
      {
        Enumerator<T> i = this.GetEnumerator();
        while (i.GetNext()) {
            T x = i.GetObject();
            x.Print();
        }
      }
}
```

The first line (and in particular the ":" symbol) indicates that *T* must support the interface IPrintable. Wherever *T* is used in the body of the class, it can be assumed to have at least this type. This is why we can call the Print function on the last line of the method without casting "x".

IPrintable is called a *bound for T* and *T* is called a *bounded type parameter*. Until now the bound on each type parameter has been Object.

Another way to do something similar without defining a new class is to add the following static, polymorphic method to the List class:

```
static void Print<T : IPrintable>(List<T> x)
  {
     for (Enumerator<T> i = x.GetEnumerator(); i.GetNext();)
        i.GetObject().Print();
  }
```

This function is parametric in one type variable *T*, which is bounded by IPrintable as before. Inside the body of the method we can access the printing behavior of the type *T*.

3.3.5 Cloneable Lists

COM+ and Java define the interface ICloneable that represents a contract to do either a "deep" (recursive) or a "shallow" (level-1) copy of an object. This is used to attach default cloning behavior to a type.

```
// The current COM+ definition of ICloneable:
interface ICloneable
{
        public Object Clone();
}
```

Using PP this can be expressed a little more precisely:

```
interface ICloneable<T>
{
    public T Clone();
}
```

T is given a precise type whenever ICloneable is implemented. Of course we normally want to clone instance of type *T* to instances of precisely the same type *T*, so *T* will normally be instantiated to the type of the class being defined. For example,.

```
final class String implements ICloneable<String>
{
    ...
    public String Clone();
    ...
}
```

We can now define extra functionality on Lists to clone an entire list. The Clone() method will be called on every element of the list. This then also becomes the default cloning behavior of our new class.

```
abstract class CloneableArrayList<T : ICloneable<T> >
                  extends ArrayList<T>
                  implements ICloneable< CloneableArrayList<T> >
{
    ...
    public CloneableArrayList<T> Clone() {
      Enumerator<T> i = this.GetEnumerator();
      CloneableArrayList<T> res = new CloneableArrayList<T>();
      int n = 0;
      while (i.GetNext()) {
          T x = i.GetObject();
          T y = x.Clone();
          res.Insert(x,n++);
      }
      return res;
    }
}
```

The first line (and in particular the ":" symbol) indicates that *T* must support the interface IColoneable<*T*>, i.e. any object of type *T* must support a method for producing another object of type *T*. Wherever *T* is used in the body of the class, it can be assumed to have at least the type ICloneable<*T*>. This is why we can call the Clone function on the last line of the method without casting "x" to ICloneable, and why the result of this operation is then known to be a *T*.

ICloneable is called a "bound" for *T* and *T* is called a "bounded type parameter." Until now the bound on each type parameter has been Object.

Another way to do something similar to the cloneable lists above is to define a new static method in some class:

```
static List<T> MyClone<T : ICloneable<T> >(List<T> x)
   { ... }
```

The method MyClone is parametric in one type variable *T*, which is bounded by ICloneable as before. Inside the body of the method we can access the cloning behavior of the type *T*, however it is defined.

3.3.6 Closures and Map

Here's the pseudo-code of an example showing how we can map an anonymous function over a list. The description of function types is greatly simplified by PP. We look at function types in more detail in Section **Error! Reference source not found.**.

```
abstract class Fun<Arg,Res> {
  public virtual Res apply(Arg x);
}
```

Here's a class for creating closures that represent the identity function:

```
class Identity<T> extends Fun<T,T> {
  public virtual T apply(T x)  //n.b. this does override apply
  {
     ldarg #x
     ret
  }
}
```

We can now add the polymorphic virtual method "Map" to the list class. We could also define this as either a static or non-virtual method.

```
class List<T> {
  ...
  public List<T2> map<T2>(Fun<T,T2> f) {
     // iterate over the list, calling f.apply(x) for each element
     // storing the results in a new list.
     ...
  }
  ...
}
```

Now we can create an instance of the Identity function, and map it over the list.

```
public static main() {
   ...
   ldloc "some List<String>"
   newobj Identity<String>();
   call List<String>::map<String>(Fun1<T,T2>)
   ...
}
```

As an aside, the method "apply" in Identity<*T*> *does* override the method in the abstract base class Fun1<*T1*,*T2*>, even though the method signatures look different. This is because, for the purposes of overriding, signature-equivalence is determined with respect to the instantiation of the class being extended. See Section 3.5 for more details.

3.4 Further Details

The previous section has introduced the proposed form of PP by examples. We now discuss the details of some of the issues raised by this design.

3.4.1 More on Instantiations

The same JITted code must work for all instantiations of a type parameter *T*. This means that restrictions are required on instantiations in order for PP to work without requiring re-JITting of code. In particular, all instantiations must have a uniform representation as far as the COM+ Runtime JIT and GC are concerned. Thus, we normally only allow a class like Enumerator<*T*> to be instantiated a boxed type, i.e. any subtype of Object.[5]

[5] Actually we later recommend that this restriction be weakened so that interfaces and fully-abstract classes can be instantiated by any types, which is very useful for describing closure-signatures. We also consider full support for instantiations by base types, with re-JITting of the relevant code, Section 6.2.

3.4.2 More on Bounded Type Variables

> *NOTE: Bounded type variables can be simulated by replacing the bound with "Object" and just using casts, and thus can be considered an extension to a simpler system that does not have them. The only reasons to implement the extension in the core of the runtime is efficiency and uniformity.*

We write a bound using "*T* : *type*" where *T* is a type variable. Most of the type parameters in the examples in the previous section were implicitly bounded by Object. Type parameters may also be unbounded, but then can only be used in very limited circumstances (see Section 3.4.5).

The verifier checks that bounds are satisfied wherever a parameterized type is instantiated. This is a completely static check.

Bounds may be defined using any type variables that are "in-scope", including all those type variables being introduced at the current class or method. Thus, bounds may be recursive, and even mutually recursive. Such bounds are useful for enforcing uniformity up to subtyping.

For example, if the definition of ICloneable is:

```
interface ICloneable<T>
{
    public T Clone();
}
```

then the typing for the MyClone method from Section 3.3.4 is

```
static List<T> MyClone<T : ICloneable<T> >(List<T> x)
   { ... }
```

Here we have a recursive bound "*T* : ICloneable<*T*>". Now, if *T* = String, then String must support ICloneable<String>, which means that it must support the method "public String Clone();". We have used the recursive constraint to force parametric types to clone themselves to their own types (actually to any subtype of their own type). This means we can be sure that the clone function will return a *T*, and we can eliminate yet another cast from the MyClone method.

There is a rather subtle point to notice here: if you can satisfy a bound with some type A, it does not necessarily follow that any subclass of A will also satisfy that bound. This is because the bound on a variable *T* may itself mention *T*, i.e. the bound may be recursive. For example, if class Point supports ICloneable<Point>, then Point will satisfy "*T* : ICloneable<*T*>". However ColoredPoint (i.e. some subclass of Point) does not automatically satisfy "*T* : ICloneable<*T*>". It will only do so if it explicitly supports ICloneable<ColoredPoint>.

3.4.3 Supertypes

Polymorphic supertypes (i.e. implemented interfaces, inherited interfaces and inherited classes) must actually be instantiated when they are mention in an "extends", or "implements" declaration. These instantiations must satisfy the relevant bounds.

3.4.4 Restrictions on the use of type parameters

"Naked" type variables may not be used to specify the class where a method, field or constructor occurs. For example:

```
static void junk<T>() {
  call T::meth()  // not allowed.
  newobj T::T()   // not allowed.  Variables may
                  // not be used to specify the method being
                  // called, just the particular
                  // instantiation of a method being called.
}
```

You can, however, access those members available at a bound:

```
class C {
    public constructor C() { }
    public virtual int meth() { ... }
}

static void junk<T : C>(T x) {
  newobj C::C()          // pushes a C, not a T
  ldarg #x
  callvirt C::meth()
  ldarg #x
  newobj List<T>::List(x)      // OK - create a singelton list
  ...
}
```

Similarly, naked formal type parameters cannot be used to specify an inherited class, inherited interface or implemented interface:

```
class junk<T> extends T ... // not allowed

class junk<T> implements T ... // not allowed

interface junk<T> extends T ... // not allowed
```

Naked formal type parameters can, of course, be used to specify the types of non-static fields, arguments and locals.

3.4.5 Unbounded type variables

Some type parameters may be completely unbounded, i.e. they may be instantiated by base types and value class types as well as reference types. We write this as "*T* unbounded". This is more general than simply writing *T* alone because the default bound is Object.

The basic rule is this: unbounded parameters may not be used in code, only in interfaces. They are only useful to define certain interfaces and abstract base classes.

For example, functions types may be described via either of the following (function types and their relation to PP are discussed in much more detail in Section 5 – in this section we just use a naive representation of them as an example).

```
abstract class Fun<T1 unbounded,T2 unbounded>  {
  public virtual abstract T2 Invoke(T1 x);
}
```

Unbounded parameters may be used to specify the signatures of abstract methods, and to instantiate other unbounded parameters. They are particularly useful with interfaces and fully abstract classes, which then get both instantiated and called at more specialized types.

Unbounded parameters may be instantiated by any type, including other unbounded parameters, base types and unboxed value class types. For example consider the following naive description closure

```
In Standard ML: (fun x -> x + y)

In COM+:
  class Closure1 extends Fun<int,int> {
    private int x;
    public constructor Closure1(int x) { this.x = x; }
    public virtual int Invoke(int y) { return x+y; }
  }

In Standard ML: (fun x -> x.Length())

In COM+:
```

```
class Closure2 extends Fun<String,int> {
  private int x;
  public constructor Closure1(int x) { this.x = x; }
  public virtual int Invoke(String y) { return x+y; }
}
```

The advantage is that fewer abstract interfaces are required to defined the space of all possible function closures. Without this, types such as IFun_I1_I1 must be created for unboxed base types, and, even worse, for anonymous functions involving value classes. In addition, later versions of COM+ may optimize aspects of the dispatch for higher order functions. Classes and interfaces constrain closures to particular shapes that may be suitable for later optimization.

Unbounded parameters may not be used within code sequences, because the code produced by the JITter will differ for different instantiations, and we require that the same JITted code work for all instantiations of the parameter. In particular, a bound of at least Object is required for:

- Parameters used to specify the type of a field of a class
- Parameters used anywhere in any declaration or instruction in a method implementation.

Basically, unbounded parameters are only useful for specifying abstract access mechanisms and identifying particular locations in *vtables* for optimized types. They must still be instantiated to more specific cases both at a call location and the call site.

> *NOTE: Unbounded type parameters can be simulated by consistent name mangling into a simpler system that just has parameterization over reference quantities.*

3.4.6 Constructors

This section is under construction.

Essentially, every constructor must be polymorphic in the sense it must be possible to use it for any instantiation of a class's type parameters. Thus you can't specify a constructor in the class List<*T*> that is only suitable for constructing List<int>.

Because of this, constructors are considered to be under the scope of the class type variables.

3.4.7 Static Fields and Methods

Static fields may *not* use formal type parameters, i.e. static fields must be monomorphic. This differs from C++ templates, and is because we don't want copies of the static variables on a per-ground-instantiation basis.

Static methods may define their own formal type parameters, but may not directly use the formal type parameters for the class. A compiler may always duplicate the class parameters for each static method.

The reason why is that static methods are simply not restricted by the type parameters to their class. Many static methods will not need all the class parameters, or will need several copies (e.g. for comparison functions), or will constrain them further, and it is simpler to make them indicate precisely which variables and bounds are required rather inheriting the bounds from their class definition environment.

When we access a static member of a polymorphic class, the specification of the member given in the IL code need not include an instantiation for the type variables of the class, since they are irrelevant.

3.4.8 Arrays

COM+ supports several kinds of arrays. Most of these correspond precisely to special kinds of parametric types with special implementations. However, Java style arrays do not quite correspond to

parametric types. This is because of their rather weird co-variant subtyping behavior where B[] <: A[] if B <: A and its associated runtime typecheck.

All other COM+ runtime array types can be treated as parametric types, and indeed the implementation of PP would often generalize existing mechanisms provided for these types.

Like all objects, arrays carry runtime type information, including a description of their parameter type. The following code *is* allowed, because the type token needed for the runtime type will be available at runtime, as explained in Section 3.5.5

```
public static mk_arr<T>(int size) {
   ldarg #size
   ldarg #size
   newobj T[,]   // some array type.
   ret
}
```

3.5 Naming, Overriding and Referencing

We must be careful to adjust the rules for referencing, unique naming, and overriding to account for PP. The rules we discuss here are sufficient for the VOS – the CLS would have to impose further restrictions, to allow sensible source-language overloading. Such additional rules are non-trivial, though many of the design choices follow from those made by GJ.

3.5.1 Referencing members

Every **call**, **calli**, **ldfld**, **stfld**, **ldsfld** and **stsfld** instruction references a *member*. The way members are referenced differs slightly in the presence of PP:

- **Without PP**: by quoting a *class*, *name* and *signature.*
- **With PP**: by quoting a *class*, *class instantiation*, *name*, *signature* and *method instantiation*. For methods in non-parametric classes, or for static members in classes, the class instantiation must be empty. For non-parametric methods the method instantiation must be empty. A method signature is quoted with respect to its *formal* parameters, not its *actual* parameters.

The actual representation of a class or method instantiation would simply be a list of type tokens, each representing a type parameter.

How instantiations appear in the concrete syntax of the assembler and the metadata of the IL are open issues. So far we have simply written out the method signature after substitution – this could be the default mechanism but a more specific mechanism could be provided when the instantiations have to be written out explicitly. The important thing is that it must be possible to know: (a) exactly which method is being called; and (b) under what the instantiation the call is happening. This allows the verifier to check that the items on the stack (e.g. the object whose method or field is being accessed) have the correct type.

3.5.2 Name-uniqueness of members

Members within classes must be "unique". The interpretation of this again differs slightly in the presence of PP:

- **Without PP**: No two members may have the same name and same signature.
- **With PP**: No two members may have the same name and signature *prior to instantiation*.

3.5.3 Determining overriding

When one class extends another, the verifier must compare method signatures in order to determine if a new virtual method overrides another method. The interpretation of this again differs slightly in the presence of PP:

- **Without PP**: We override any method with exactly the same name and same signature. There can only ever be one such method.

- **With PP**: We override any method whose signature is the same, taking into account the instantiation of the class we are declared to extend. Thus, to determine overriding, we compare signatures *after* instantiation. *This method must not be ambiguous*.

The following indicates why we must be careful when defining overriding:

```
class A<T> {
   virtual void foo(T x) { ... }
   virtual void foo(String x) { ... }
}

class B extends A<String> {
   virtual void foo(String x) { ... }
}
```

Which method does "B::foo" override? The comparison is ambiguous, after the instantiation for A is taken into account. Thus the class B because the overriding is ambiguous.

A similar problem occurs when combining interfaces using "extends".

3.5.4 Determining implementation

When a class claims to implement a certain interface, the verifier must again compare signatures to see if this is really the case. This is resolved in the same way as overriding.

3.5.5 "requires" annotations

Section 3.6 describes how some methods require runtime type data for type variables. This is so runtime type data can be correctly constructed for instances of polymorphic classes.

For example, the method "goo" might be declared as:

```
public global static goo<T1>() requires T1 { ... }
```

This means that an extra, hidden parameter representing the type *T1* must be passed at runtime. COM+ looks after the passing of these values. However, they form part of the contract of calling the procedure, and are part of the method signature. If no "requires" declaration is given then it is assumed that *all* the type variables must be passed. This may be less efficient.

Because these declarations are part of the signature of a method, they are relevant when determining name-uniqueness and overriding.

3.6 Runtime Typing

A critical design issue for PP is runtime type information, i.e. whether the runtime type data stored for an object should indicate the instantiation of its class's type variables, and if so, how this information is created, stored and passed around to the places where it is needed. The information is essentially static, i.e. it records a fact that is known at verification time.

Do we store type instantiations at runtime? The issues are:

- *Type safety*. In the in the presence of reflection, casting, marshalling/remoting, and versioning, some kind of checks are needed to ensure type safety. This does not necessarily mean that runtime type information must be stored, but it seems the best option.

- *Efficiency*. The checks required for type safety in the presence of casting etc., will always have some kind of runtime cost.

- *Coherence with Java-style arrays*. Java-style arrays (i.e. with a runtime type parameter and covariant runtime typing, with a typecheck on every write) are, for better or worse, the default in COM+. These clearly possess a kind of polymorphism, in the sense that the array type constructor is polymorphic. The type parameter may be inspected at runtime. Will languages targeting COM+ sometimes require exactly the same behavior from other polymorphic structures?

- *Coherence with covariance of Java-style arrays*. This is really a side issue, but is mentioned here because the whole reason why Java-style arrays require the storage of the runtime type parameter is because Java supports *co-variant* subtyping for these arrays. Thus the COM+ *must* store the array type parameter, because we have to be able to perform a runtime typecheck against this parameter every time write to an array. (Alternatively, we could do a runtime type check everytime we do a read from an array, but this would be less efficient).

- *Coherence with OO paradigm.* Given the presence of runtime typing in the system as it stands, will users naturally expect runtime type information for parameters as well.

3.6.1 Casting & Type Safety

Permitting safe casting from Object to particular List types is the primary cause of the type safety problem. Two important uses of casting are:

- casting after reading in serialized data.

- casting from Object when interfacing to APIs that don't support parametric polymorphism, but store and handle values via the type Object.

Four solutions are available to the type safety property:

1. Do not store type parameters, but perform a typecheck/cast everytime we *access* a polymorphic field. Compatible with contravariant subtyping.

2. Do not store type parameters, but perform a typecheck/cast everytime we *write* to a polymorphic field. Compatible with covariant subtyping.

3. Store type parameters, with no code duplication. Type tokens must be stored and passed, as discussed below, with a possible runtime cost. A typecheck is only required when, for example, casting from Object to List<String>. Not immediately compatible with either covariant or contravariant subtyping, unless further casts/typechecks are inserted.

4. Perform full code expansion for every ground instance. Major code expansion, but most efficient code, when the impact of the increase in the working set is ignored.

This proposal proposes that values representing the type parameters are available at runtime, via option 3.

Runtime type information involves a cost. For example, the runtime will have to generate some information on a per-instantiation basis, e.g. tables for both List<String> and List<Object>, and should share this information between all objects that have these types.

The costs of runtime type information can manifest themselves in terms of either speed or memory, i.e. there is a space-time tradeoff between different representations, as documented below in Section 0. It is not clear whether the cost associated with runtime type information is palatable in any of its forms. Before we commit to this design, we propose that experiments be done to measure the costs of passing and storing runtime type tokens. It is possible that this cost could provide reasonable grounds for rejecting PP. However, in the opinion of the authors the costs will be probably prove negligible in comparison to existing overheads.

Note that we believe option 3 will have the best space-time performance of any of the other options above, and subsequently should also give better performance than the JVM when compiling polymorphic versions of Java such as GJ.

> *Why can't we get away without checks or runtime type parameters? Consider the following attempts:*
>
> Q. How about just allowing casts from Object to List<Object> as long as the object was actually some List<T>, since, after all, the T in List<T> is bounded by Object?
>
> No. *We cannot treat all lists as List<Object>. because then we could cast a List<String> to Object, then List<Object>, set its head to an Object and break the type system. That is, because the head of a list is mutable, the* T *in List*<T> *must be* non-variant, *unless we have runtime type checks as for Java-style arrays.*
>
> *In other words, when we say "the* T *in List*<T> *is bounded by Object", we mean it is bounded* statically. *That is, we can only instantiate List*<T> *with subtypes of Object, but that's not the same thing as saying all values of Object can be used wherever we have an* T.
>
> Q. How about allowing unchecked casts from List<B> to List<A> when B is a subclass of A?
>
> No. *Again, this doesn't work because* T *must be non-variant in List*<T>.
>
> Q. Couldn't we contemplate some kind of super-cast that checks every element is a String when casting Object to List<String>, and then do away with type parameters at runtime?
>
> No. *Variance is again the problem: say we have an object handle with type List<Object>. We then super-cast to List<String>, because every element is indeed an String. We then modify the head of the list* using the original handle *(which we duplicated), putting an Object in the head. This breaks type soundness.*

3.6.2 Ground Instantiations

A *ground instantiation* is a type containing no type variables, e.g. List<String>, but not List<List<*T*>>.

Every time we instantiate a polymorphic type with ground types, this can give rise to a whole series of ground instantiations of related types. This is called the *closure of ground instantiations.* Those familiar with C++ will recognize this as the set of templates that are compiled by a link-time template expansion compiler.

Ground instantiations are crucial at runtime because of the following very important observation: *no objects are actually polymorphic at runtime. That is, every object is created with some particular values for its class type parameters.* The only exception to this rule is for polymorphic delegates, which are discussed in Section XXX.

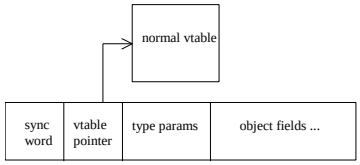
An important aim of this proposal is to make it possible to remove all ground-instantiation data from a COM+ assembly, and leave it to the COM+ Runtime to create any necessary ground-instantiation data.

3.6.3 How are actual type parameters stored at runtime?

> *This section needs to be updated to reflect a more complete understanding of how arrays are currently represented in the heap of the runtime. In particular, it seems the lower order bit of the runtime type token for an array object indicates that the object is indeed an array, and that the runtime type token actually points to a structure like that in Option 4 below. We can use precisely the same representation for all parametric types.*
>
> *In addition, it may be possible to be even more efficient in the case that the type parameter is minimal, e.g. List<Object> and Object[]. In this case, we could set the lower order bit to 0, i.e. pretend the object is monomorphic, and then the RTT would point directly to the vtable. (This is just an idea – it would mean that there is almost no loss of efficiency at all for existing code due to polymorphism)*

Given we must store actual type parameters for instances of polymorphic classes, then where does this information get stored?[6] The information will certainly be some kind of token per parameter. Note that when we say "tokens for types" we mean "handles to runtime types", which really means handles to specifications of ground instantiations of types.

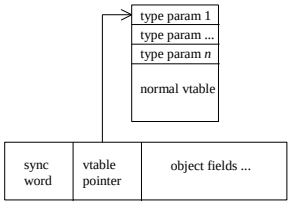
- Option 1: Store the tokens as hidden private fields in the object. The tokens are available to the constructor, as they must be for any of these options.



Good for speed, not so good for space. Creating polymorphic objects is very simple and fast, because we just write the tokens directly into the runtime representation, but polymorphic objects take extra space, e.g. a small polymorphic object such as ListItem<T> in a linked list implementation could take 4 words instead of 3. If there are relatively few small polymorphic objects this may be the best solution.

If the space performance of the above option is not acceptable, then per-ground-instantiation data seems pretty inevitable (not per-instance, but per-ground-instantiation). Then the runtime looks after the creation of this data every time a new ground instantiation of a class is specified. New ground instantiations can either be detected on the fly in constructors, or as classes are loaded.

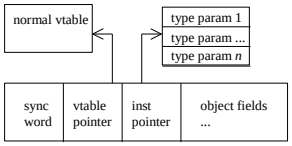
- Option 2: Store the tokens in the vtable for the object. This means a new copy of the vtable for each ground instantiation.



Reasonable for speed, reasonable for space. Creating polymorphic objects requires a table-lookup of some kind based on the type parameter, as we must find the correct vtable for the given parameter. Polymorphic objects take no extra space, but the vtable contains duplicated information. Again the JIT could optimize in special cases.

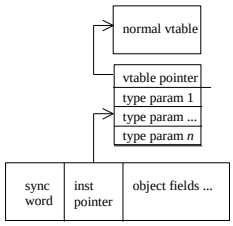
[6] This discussion assumes a model of object layout that may not be accurate, since the layout of objects and object handles is not explicitly described in the COM+ specifications.

- Option 3: Have another slot in each parametric object allocated to point to the per-ground-instantiation data.



Reasonable for speed, Not so good for space. Creating polymorphic objects again requires a table-lookup of some kind based on the type parameter. Polymorphic objects take one extra word, which since we are concerned most about many small objects is probably not acceptable. Per-ground-instantiation data is kept to a minimum.

- Option 4: Indirect the vtable via per-ground-instantiation data.



Poor for speed, Excellent for space. Dynamic dispatch for polymorphic instances becomes significantly slower, though this may not be a disaster. Space usage is kept to a minimum. Some type tokens must still be passed around at runtime.

- Non-option 1: Duplicate code for every ground instantiation. Excellent for local speed, Excellent for per-object space, Pathetic for both once code size is considered. No per-instance data is needed, runtime type tokens are not required.

Options 1 and 2 look best. Probably the best place for the information is in the vtable for the object (option 2). That is, vtables contains per-instantiation data, which means vtables must be duplicated on a per-instantiation basis. It is up to the runtime to create, share and track this data – indeed along with verification this is the essence of the service provided by the runtime with regard to PP. Other options are possible, but duplicating the vtables is likely to be the best:

Final types that implement polymorphic interfaces may optimize the storage of their runtime type information (i.e. no runtime type information need be stored at all). e.g.

```
final class Integer4 implements IComparable<Integer4> { ... }
```

The only code that could possibly require the runtime type parameter from the runtime format for this object is contained within this class, where the appropriate token for Integer4 is already available.

3.6.4 Where are type tokens needed?

Ultimately, tokens for types must be available whenever we (a) construct a new instance of a polymorphic class using **newobj** or (b) use a type variable as part of a **cast** or **ldtoken** instruction. This means that when we execute

```
newobj List<T>()
```

we must have access the type token *T*. The token is needed to create or find the per-ground-instantiation data for List<*T*>. This data is then written into the object, according to the representation chosen in the previous section.

3.6.5 Getting type tokens to where they're needed

Type tokens must be passed down to methods that require them. Not all parametric methods require them, so they are only needed to use **newobj** on a type involving the tokens, use **call** or **callvirt** on a method that itself requires them; or to do a **cast**, **box**, **unbox**, etc. on a type involving a type parameter.

Thus, *methods must explicitly declare in their signature which runtime type parameters are required for the execution of the method*. Polymorphic virtual methods should declare all their runtime type parameters in this way, because an overriding version of the method may require these type tokens. Some tokens may be accessed via the "this" pointer, i.e. via the runtime type data for the object for which the method is being executed. [7] Thus, virtual and non-static, non-virtual methods need not specify that they require these tokens.

We write the signature of a method that requires a runtime type token in the following way:

```
public static List<T> singleton<T>(T x) requires T
```

If the JIT or some global analysis tool can determine that *T* can be derived from some other source, then these declarations could be ignored or removed. The declarations are certainly verifiable – they just determine which tokens are actually in scope where for the use of code that requires them.

To summarize, where do tokens come from? It depends on what kind of code we're executing:

- in a virtual or non-static, non-virtual method: class type parameters get accessed via "this" pointer, method type parameters are passed implicitly by the JIT as arguments
- in a constructor: required type parameters are passed implicitly by the JIT as arguments
- in static call: required type parameters are passed implicitly by the JIT as arguments
- for **newobj, cast** etc.: type tokens for the type parameters used in these instructions must be made available. These instructions may generate new per-ground-instantiation data for the new ground instances implicit in the instruction.

The JIT takes responsibility for constructing, managing and passing type tokens.

3.6.6 Examples of where type tokens are required

Consider the type Pair<T>:

```
class Pair<T> {
  private T x,y;
  public constructor Pair(T x, T y) {
     this.x = x; this.y = y;
 }
}
```

The above constructor does *not* require a token for *T* at runtime, because it does nothing tricky, e.g. constructs no new objects. However, whenever a **newobj** uses this constructor, a runtime type token will be needed.

```
global static foo() {
   ldstr "String 1"
   ldstr "String 2"
   newobj Pair<String>::Pair(T, T)
}
```

The "foo" method cannot require any runtime tokens, because it is not polymorphic. However, per-instantiation data for the ground instantiation Pair<String> is needed for the **newobj** instruction. This

[7] Only invariant type parameters may be accessed in this way – see Section 6.1 on the proposal for variance. For the moment we'll assume all parameters are invariant.

would be created on-the-fly if it has not already been created. This will effect the JITted code for the **newobj** instruction.

```
global static goo<T>() requires T {
   ldnull
   ldnull
   newobj Pair<T>::Pair(T, T)
   ...
}
```

The "goo" method requires the token *T*, because the **newobj** instruction may need to create per-instantiation data for different ground instantiations for *T*. Even though both the fields are **null**, objects created by "goo" still record the instantiation of *T* in their runtime type data.

```
global static hoo<T>(...) requires T {
   call goo<List<T>>()
   ...
}
```

The "hoo" method clearly requires the token *T*, in order to be able to call the "goo" function. The JITted code for the **call** instruction may need to create the token for the ground instantiation List<*T*> which needs to be passed to foo.

```
global static koo<T>() requires T {
   ldnull
   ldnull
   newobj Pair<List<T>>::Pair(List<T>, List<T>)
   ...
}
```

As for "goo", the "koo" method requires the token *T*, because the **newobj** instruction may need to create per-instantiation data on the fly for different ground instantiations of List<*T*>.

```
public global static loo<T>(Object x) requires T {
   ldarg #x
   cast Pair<List<T>>
   ...
}
```

The "loo" method attempts a different cast depending on the given type parameter. The **cast** instruction will have to create the per-instantiation runtime type token for different ground instantiations of List<*T*> on the fly. This data will be the basis for implementing the runtime casting check.

```
public global static moo<T>(Object x) requires T {
   ldarg #x
   ldc 9
   ldc 9
   newobj T[,] // i.e. create a 9x9 array
   ...
}
```

The "moo" method requires *T* because the array it creates must contain the appropriate runtime type token in its type descriptor.

3.7 Object Marshaling, Remoting and Versioning

This discussion assumes a model of remoting/serialization/marshalling/versioning something like Java's, since the design docs for these aren't really clear as to what's going on, especially with respect to stored type information and type soundness. Thus it may need to be changed depending on the COM+ design for these features.

Question 1. Is there any interaction between marshalling, versioning and PP?

Yes. When we do a read of some data, we receive a reference of static type Object. The remoting facilities ensure that the runtime type tags for the data are correctly constructed and that the data corresponds to these runtime tags. This allows us to later cast to the type that we "know" the data has, as long as casting allows us to do this sort of thing.

Versioning means that the serialized/remoted representation of an object may be compatible with many different versions of a class, e.g. if we add a non-remoted field to a class. Thus, we would expect that under certain circumstances the serialized version of List<A> may be compatible with List'<A>, or List<A> with List<A'>, where the ' represents a new version of the class, and we assume List' is versioning-compatible with List', and A' with A.

Question 2. So can we marshal parametric types such as List<String>?

Yes. It seems paramount that parametric storage types be remotable and versionable in the most generous way possible.

Question 3. Can a marshaled version of List<A> be read in and used as a List<B> if every element is indeed a B at read-time?

No. In principle this may be OK, if marshalling has really taken a copy of the list, but the implementation mechanism described below would rule this out, because the runtime tags created when reading the object would record it as being a List<A>.

Question 4. Are the type parameters stored in the external representation of remoted data?

Yes. We could contemplate some read-and-check for marshaled data if we knew the type we were expecting when doing a read, but the marshalling design does not allow this, since the marshaler returns an Object upon reading, and does not have access to the static type expected.

Question 5. But couldn't we work out the type parameters as we read the data?

No, not really. We could compute a bound ("this object can be cast once to any List<B> where B <: A"), and store this bound with the object, and modify it after the cast-once. But this sucks.

Question 6. So type parameters are specified and checked by the cast after the read?

Yes.

3.8 Value Classes

Polymorphic value classes are allowed. Here are the outlines for two value class examples:

```
valueclass Pair<T1,T2> {
  public T1 x1;
  public T2 x2;
}

valueclass PrintablePair< T : Printable<T> > {
  private T x; T y;
  public PrintablePair(T x, T y) { ... }
}
```

Runtime types are not stored for unboxed value class instances. The verifier ensures we don't try to query the runtime type of an unboxed value classes. If we box a value class then the verifier annotates the result with the appropriate runtime type information. This may mean generating new runtime type tokens for the ground instantiations that correspond to the particular value class instance.

3.9 Related Issues

3.9.1 Representation in COM+ Metadata

How are parametric types represented in COM+ metadata? Essentially, all the declarations that we have shown must be mapped down to the data attached to type and method definition tokens in the COM+ metadata structures. For example, this means every class will have a list of type variables associated with it. For non-parametric classes this list will be empty. Similarly, it will be possible to specify a type using a type variable – this will simply be a token that resolves to a name.

TBD: Concrete syntax in the Assembler.

3.9.2 Inlining

The proposed design fully supports the inlining of polymorphic methods by the JIT.

3.9.3 Class Objects

Reflection allows us to dynamically query the class of an object, and to ask things about that class via Class objects. The class objects that appear at runtime should reveal type instantiations.

3.9.4 Reflection

Reflection allows us to dynamically create classes. PP will impact on this API, but in a straightforward way. The current Reflection design should be checked to make sure it is backward compatible with PP.

3.9.5 Debugging

Ground type instantiations should appear in the debugger whenever they are available at runtime.

3.9.6 COM interop

Any issues here?

4 Covariant Return Types

Covariant return types have been proposed for Java and are a straightforward relaxation of the typing rules for the VOS. [8]

A covariant return type is when an overriding method declares a more specific return type than that of the method it overrides. For example:

```
class List {
   virtual List Clone() {...}
}

class ArrayList extends List {
   overrides ArrayList Clone() {...}
}
```

Because ArrayList is a subclass of List, this is a covariant return type. (The phrase "covariant return type" means the return type varies in the same direction as the subclass-super class relationship: ArrayList < List, just as String < Object. In the example above, if ArrayList were not a subclass of List, the return type would not be covariant, and the code would be just as illegal as it is today.)

When you call a method in a language supporting covariant return types, the compiler considers the compile-time type of the object being invoked. In the example above:

```
List l = new ArrayList();
l.Clone();  // return type is List
ArrayList al = new ArrayList();
al.Clone(); // return type is ArrayList
```

You can think of this as the compiler automatically casting the return value to String when appropriate. However, this is an unusual cast, because it can never fail.

Note that I am recommending covariant return types, not covariant parameters. If subclasses could demand more specific inputs than the super class, what would happen when you invoked the method on an object that had a compile-time type of the super class, but at runtime turned out to be the subclass?

4.1.1 Uses

The example which sparked this is that WFC has a ListBox class with a read-only Items property that returns a ItemCollection. There's also a subclass of ListBox called CheckedListBox. It wants to add a few methods to ItemCollection (additional "Add" methods). CheckedListBox can return a MyItemCollection instead of a normal ItemCollection, but users will have to cast it to MyItemCollection in order to use the new methods.

```
class ListBox extends Control {
  public ItemCollection Items {
    get { return new ItemCollection(...); }
  }
  public static class ItemCollection { .... }
}

class CheckedListBox extends ListBox {
  public ListBox.ItemCollection Items {
    get { return new MyItemCollection(...); }
  }
  public static class MyItemCollection extends ListBox.ItemCollection {
    public int Add(Object value, int checked) { .... }
  }
```

[8] This description is mostly due to Nick Kramer.

```
}
```

With covariant return types, CheckedListBox.Items can declare its true return type:

```
class CheckedListBox extends ListBox {
  public MyItemCollection Items {
    get { return new MyItemCollection(...); }
  }
  public static class MyItemCollection extends ListBox.ItemCollection {
    public int Add(Object value, int checked) { .... }
  }
}
```

One of the problems with our current collections properties proposal is that there is no useful implementation to inherit from, because if there were an implementation, you couldn't get the types right. With covariant return types, that's no longer true.

4.1.2 Properties

Properties are really just pairs of methods with syntactic sugar for calling them. We saw above that Get methods work find, but Set methods are problematic (the covariant parameter issue). The easiest solution is to say that read-only properties can be covariant, but read-write properties may not be.

4.1.3 Cost/Benefit

- Covariant return types are well grounded in type theory
- They are more complex language and compiler
- Class libraries and WFC are easier to use (reduced casting, more type safety)
- Users must understand that we are over**riding** an existing method, not over**loading** based on return type. In languages such as VB and Safe-C which require the "overrides" keyword, this is not as much of an issue.
- They are much easier to understand than other type system extensions like parameterized types (a.k.a. templates, generics) and virtual types, though these other extensions are compatible with covariant return types.
- Today, although no language supports it, the execution engine allows you to overload methods based on return type. In the absence of explicit execution engine support for parameterized types, a compiler can fake parameterized types by overloading things based on return type and inserting explicit casts as appropriate. If we implement covariant return types in the obvious way, that can no longer be done. (We could still add explicit support for parameterized types, however)

5 Generalized Delegates

It has long been recognized that *function types* are a very useful addition to a type system. At the level of a runtime machine, values belonging to function types are called *closures*. The crucial design issues are *how we describe function types* in IL metadata, *how we describe closure templates in IL metadata*, *how we make new closures* and *how we represent closures at runtime*.

Function types are essentially *structured delegate types* and we are essentially proposing an extension to Microsoft's existing *named delegate types. C*losures simply become *delegate objects*. Typically a closure will itself be the *delegate recipient* associated with the delegate object.

Delegates as they stand are *almost* adequate to use for function types and closures in source languages. The primary problem with delegate types is that they are individually named, rather than belonging to a general space of types where compatibility is determined structurally.[9] Each new delegate type must be defined by a new class, which generates a new type name. This greatly reduces their utility, because delegate types in different unrelated components can never be compatible with each other. This is a typical problem that arises from relying too heavily on named types in a type system, and it certainly rules out the use of delegates as they stand for existing functional languages. Microsoft should seize the initiative and generalize the hard work they have already done on delegates, in order to allow the efficient implementation and flexible reuse of function types and closures for functional programming languages.

For those more familiar with either function types or delegates, the following guide may be helpful:

Delegates	**Function Types**
Delegate types, i.e. the named types associated with delegate classes	*Function types*
Delegate classes	*Closure templates* or *syntactic closures*
Delegate objects with an associated *delegate recipient*	*Closures* where the environment has been separated out to be another object in the heap
Delegate objects which are their own delegate recipient	*Closures* where the environment is stored in the closure itself
Multicast delegates	No real counterpart.

Our proposals are as follows:

1. (Required) We propose that delegate types be defined in PE files by structure, rather than by defining new classes (we will use new classes to define implementations of delegate types). Array types are already defined structurally in COM+, i.e. individual array types are not named. Delegate types should be represented in exactly the same way. We will flesh out the ramifications of this requirement with the COM+ team.

2. (Required) The COM+ Runtime support *structural type equivalence* for delegate types, rather than the existing *named type equivalence*. Delegate types should thus be given a similar status to the existing array types, i.e. two delegate types are equivalent if they have their Invoke methods have the same type.

[9] Actually, it is not entirely clear from the COM+ delegates specs how exactly delegate types are represented in the COM+ Runtime. We will follow up on this point with the COM+ team.

3. (Required) It is the responsibility of the COM+ Runtime to assign unique runtime type descriptors for each delegate type.

4. (Required) The COM+ Runtime must support very light weight delegate classes, in order to reduce the inherent cost-per-class for a delegate class to, approximately, less than 100 bytes in a PE file and less than 200 bytes at runtime. Our measurements indicate current overheads of at least 170 and 640 bytes respectively.

5. (Option) Cool, the CLS, VC++ and VB should strongly consider adding structured delegate types to their languages.

6. (Option) Support co/contra variant subtyping for delegate types. This is trivial to implement inn the verifier. It need not go in the CLS.

7. (Future Work) MSR and the COM+ Runtime will together stress test the use of delegates for closures to ensure that they are sufficiently efficient for the purposes of functional programming languages.

8. (Future Work) MSR will contribute a proposal to extend the delegates mechanism to support the fast invocation of curried functions. This is essential for the efficient implementation of many functional programming languages.

9. (Future Work) MSR will contribute a proposal to support truly polymorphic delegates, presuming that some parts of the PP proposal of Section 3 are adopted.

10. (Future Work) MSR will eventually contribute a proposal to support typesafe ways for delegate objects to modify their code pointer and recipient fields in order to provide some rudimentary support for thunks. MSR will test the adequacy of this representation by compiling Haskell to it.

5.1 How we got to this design

It was not immediately obvious to us that delegates were suitable representations for function types and closures. We shall use the following examples to demonstrate the ramifications of various design choices:

(A) Function Type: `(String, String) -> int`

Closure Template: `(λ(s1,s2). fv1.length + fv2 + s1.length + s2.length)`
Free Variables of Closure Template:
```
String fv1
int fv2
```

(B) Function Type: `int -> int`

Closure Template: `(λi. i + fv1)`
Free Variables of Closure Template:
```
int fv1
```

"`(λ...)`" is used to represent *closure templates* (i.e. syntactic closures) in a programming language such as SML or Haskell. In COM+ we propose that a closure template be mapped to a delegate class.

A closure is an instance of some closure template, and different closures will have different values for the free variables of the closure template. There may be many instances of each closure template, and many closure templates for each function type.

Some requirements:

1. Function types are subtypes of Object, or at least can be via boxed in order to allow this. This means we can store members of function types in the standard collection classes.

2. Function types are not named , i.e. they utilize structural equivalence.

3. Closures must carry sufficient runtime type information that we can safely cast from Object to function types. We do not need to be able to cast to a particular closure template, i.e. we do not have to be able to discover the template that the closure originated from. However, for GC purposes a closure will have to store some data specific to the closure template.

4. Efficient application/invocation.

5. Different function types depending on how many arguments get consumed in a particular application.

6. The overhead to describe a function type is very low.

7. The overhead to describe a closure template is very low. In particular, we would like to avoid having to define a new class for every closure template. In most naive encodings a new class is required to hold the free variables of each closure template.

8. The runtime cost of creating a new closure is very low.

9. Closures are stored in the heap.

10. The overhead to represent a closure at runtime is very low.

To quote from the experiences of someone (Gerd Stolpmann) who tried to translate OCaml into Java bytecode:

> *At the first glance, the direct translation from Ocaml to Java bytecode seems to be a superior approach, as one of two interpretation steps can be avoided. But Java bytecode is designed to represent Java that does not know some of the core features of Ocaml, namely closures [... and the automatic elimination of tail recursions]. Because of that closures would have to be represented by classes, and this means that even small OCaml programs would be translated to hundreds of classes.*

5.1.1 Describing Function Types

Objects are more general than functions/closures, in the sense that a function type corresponds quite closely to a method dictionary (vtable) with only one entry. We can exploit this to describe function types via *abstract base classes*. If we were prepared to do a lot of casting, then one simple class might be sufficient

```
abstract class Delegate {
  public Object Invoke(Object x);
}
```

However delegate types are naturally *parametric*. Just as arrays can be thought of as simple, built-in parametric types, one way to describe function types is by using parametric abstract base classes.[10] For one-argument functions a suitable type is:

```
abstract class Delegate<Arg,Ret> {
  public Ret Invoke(Arg x);
}
```

[10] Using these "parametric" descriptors does not mean the whole parametricity mechanism described elsewhere in this document would need to be implemented: here we are just using his notation to help our thinking, and to show how the two mechanisms work together.

The type parameters are unbounded, in the sense of Section 3.4.5, which means they can be instantiated by base types, value classes as well as reference types.

At the level of a runtime system, we really need many different delegate types, depending on the number of arguments that get consumed by an application of a delegate. Thus we need a *family* of parametric abstract base classes. In COM+ we would implement this as a single parametric type constructor taking a variable number of arguments.

```
// The built-in class of structured delegate types
abstract class Delegate<Ret, Arg1, ..., ArgN> {
  public abstract Ret Invoke(Arg1 x1, ..., ArgN xN);
}
```

5.1.2 Describing Syntactic Closures

Logically, a syntactic closure is a subclass of an instantiation of the class Delegate<*Arg1, ... , ArgN, Ret*>. For the moment we will assume that each closure template specifies values for *Ret, Arg1, ..., ArgN*, and we restrict ourselves to *monomorphic* closures, i.e. closures where concrete ground instantiations are given for each of these type parameters at the time the closure template is defined (polymorphic functions are described in Section **Error! Reference source not found.**).

The "free variables" of a closure become fields of the new class, and are used when the function is applied. For example, consider Example A. The closure template can be represented as:

```
class ExampleA extends Delegate<String,String,int> {
   private String fv1;
   private int fv2;
   public int Invoke(String s1, String s2)
        { return (fv1.length + fv2 + s1.length + s2.length); }
   public constructor MyClosure(String fv1, int fv2)
        { this.fv1 = fv1;
          this.fv2 = fv2; }
}
```

This representation is *logically* adequate (at least for monomorphic functions), but may not be *pragmatically* adequate:

- There is an extra indirection upon function application. Typical implementations of closures don't have to indirect via the object's vtable. (Note: experience from COM+-ML indicates that virtual method application is very fast in JITted COM+, thus this is not a major problem)

- A new class must be defined for each syntactic closure. Languages like SML and Haskell generate thousands of closures. These leads to prohibitive space usage both in the executables and in the working set. Experiments indicate that each class uses approximately 170 bytes in an executable and 640 bytes at runtime. This is too much considering that thousands of syntactic closures may be defined even in very small programs.

However, this is a good starting point. We must now decide:

- If syntactic closures are *actually* represented like this in PE files. Certainly, the cost of doing so is prohibitive for functional languages.

- If closures are *actually* represented by the object layout implicit in the above class. This depends on (a) whether the runtime can prevent users defining their own subclasses of "Fun" (i.e. whether the runtime can optimize closures at all); and (b) whether it is thought worthwhile to optimize their representation to something smaller or faster. (a) is already required for Delegates, so is not a major issue.

5.1.3 Thunks

This section is work in progress...

Thunks are modifiable closures where the code that gets executed when the closure is re-evaluated also differs on later calls. Thunks are effectively *overwriteable values.* Our requirements for thunks are:

1. Allow closures to represent thunks, i.e. some support for self-mutating closures.

2. Allow for the possibility that thunks can be shorted out at GC time.

6 Enhanced Parametric Polymorphism

6.1 Variance

This section is not yet complete. We propose that the runtime allow the user to declare variance for type variables, and to check the validity of this variance each time a variable is used when determining subtyping.

6.2 Base type instantiations

This section is not yet complete. We propose that COM+ allow parametric classes to be instantiated by base types and value class types, and that it re-JITs the code for each such instantiation.

A Type System for Parametric Polymorphism

A.1 Basic Type System

The purpose of this section is to document the design of the proposed PP mechanism in detail to enable further discussion in the COM+ team. The description does not yet include privacy annotations, abstract annotations, or any of the extensions listed in Section 3.

A.1.1 Notation

[*item*] = a list of *item*s.

A.1.2 Types

A *type variable* is some kind of unique identifier (e.g. a token). We will denote type variables using α, β etc.

Class, interface or *value class identifiers* are simply strings. These are denoted by C, I and VC respectively, or simply T when it doesn't matter which kind is being used.

A *type* is a member of the following grammar:

type =

T	(static aspects of a defined type)
| T < $\tau_{1, ..., } \tau_n$ >	(polymorphic aspects of a defined type)
| α	(type variable, which must be in scope)
| **I1**, **I2**, **I4**, **I8**, **R**, **R4** etc.	(unboxed base types)

Not all types are make sense, so wellformed types are defined below.

$\tau_{1, ..., } \tau_n$ in "T < $\tau_{1, ..., } \tau_n$ >" are called the formal type parameters for a constructed type.

Equality on types is by structure, so one type List<String> is considered identical to any other, regardless of the representations used for each.

A.1.3 Type Definitions

A *class definition* is of the following form. We only give the concrete syntax, since its just as clear as the corresponding abstract syntax:

```
class class-identifier<α1 : bound1,
                       ...,
                       αn : boundn>
        extends extends
        implements implements1, ... implementsn
{
      constructors
      methods
      fields
      class-constructors
      static-fields
      static-methods
 }
```

α_1, ..., α_n are called the formal type parameters for the class. The scope of α_1, ..., α_n is everything excluding the class constructors and static members. We omit privacy, abstract, package and final declarations. The items occurring inside a class definition are defined below.

An *interface definition* is of the form:

```
interface interface-identifier<α1 : bound1,
                               ...,
                               αn : boundn>
        extends extends1, ... extendsn
{
      methods
}
```

and a *value class definition* is of the form:

```
valueclass valueclass-identifier<α1 : bound1,
                                 ...,
                                 αn : boundn>
{
      constructors
      methods
      fields
      class-constructors
      static-fields
      static-methods
}
```

The scope of type variables is defined as above. We assume the *type environment* contains a list of all the available type definitions.

A.1.4 Arrays

There is a special class definition for each array type that is supported by the system. For example, we will assume one dimensional arrays have been defined by

```
class Array<α : Object> { }
```

We continue to write these array types as α[].

A.1.5 Method and Field Definitions

For the purposes of this section we ignore the question of how external references to methods and types are specified.

A.1.5.1 Method Defs and Uses

Concrete syntax:

return-type
method-identifier<α_1 : $bound_1$,
...,
α_n : $bound_n$>(*arg*$_1$, ..., *arg*$_n$)
requires $requires_1$, ..., $requires_n$
{
method_impl (optional)
}

Abstract syntax:

method_def =
∀*tyvars* ≤ *bounds*.
arg-types: [type],
return-type: type,
requires: [tyvar] (type variables requiring runtime type tokens)
...
impl: method_impl (optional)

method_use = *type*::*method_def*<*[type]*>

A.1.5.2 Fields

field_def =
name: string,
type: type

field_use = *type*::*field_def*

A.1.5.3 Method Implementations

method_impl =
locals: [type],
code: [inst],
ExceptionTable,
...

inst =
| **call** *method_use*
| **callvirt** *method_use*
| **newobj** *method_use*
| **newarr** *type*

| **ldfld** *field_use*
| **stfld** *field_use*

| **ldsfld** *field_use*
| **stsfld** *field_use*

| **instanceof** *type*
| **cast** *type*

| **box** *type*
| **unbox**

| **ldtoken** *type*
| ...

A.1.6 Verification

A.1.6.1 Scope

The *type variables in scope* are defined implicitly by the indentation structure of the syntax above.

The *bound of α in the current scope* is the bound declared for α when it was declared.

A.1.6.2 Substitution

Simultaneous substitution of *type'*s for *tyvar'*s is defined in the obvious way over structures such as *type* and *[type]*, and written *ty*[*types/tyvars*].

A.1.6.3 Subtyping

τ_1 is a subtype of τ_2 according to the reflexive, transitive closure of the following rules:

$$\frac{}{\alpha \le \text{the bound of } \alpha \text{ in the current scope}}$$

$$\frac{\tau_1 \le \tau_2}{\tau_1[] \le \tau_2[]}$$

$$\frac{}{C\langle tys\rangle \le C.extends[tys/C.formals]} \qquad \frac{i \text{ in } 0...\#C.implements}{C\langle tys\rangle \le C.implements_i[tys/C.formals]}$$

$$\frac{i \text{ in } 0...\#I.extends}{I\langle tys\rangle \le I.extends_i[tys/I.formals]}$$

Note that the absence of a variance rule means no variance of type parameters is allowed, apart from arrays.

A.1.6.4 Overriding

Overriding in COM+ is based on the "virtual" flags and on signature comparison. In the presence of PP, the usual overriding rules are used, except that signature comparison is performed *after* substituting the instantiation specified for the base class through the description of the base class. (The actual implementation would not substitute, but would carry an explicit assignment of types to type variables).

A.1.6.5 Name uniqueness

See Section 3.5.2.

A.1.6.6 Satisfying Bounds

The instantiation *C*<*actuals*> "satisfies the bounds*"* if

- #*actuals* = #C.formals*;* and
- for each *i* where a bound is defined in *C.bounds*, $actuals_i \le C.bounds_i$ [*actuals*/*C.formals*]

Similarly for interfaces and value classes.

A.1.6.7 Availability and Requires

The *type variables required at runtime* by an instruction are precisely:

- All variables used in the *type* parameter to a **ldtoken**, **cast**, **instanceof**, **newobj** or **newarr** or **box** instruction.

- All variables used in those types corresponding to the required type variables of a **call, callvirt** function.

The *type variables available at runtime* in a method body are precisely:

- Those variables specified as "required" in the method header.

- Non-variant variables that are parameters to the class, as long as the method is non-static.

A.1.6.8 Checks

The checks required are:

- All uses of type variables must refer to variables in scope.

- All instantiations of type variables in a *type* or *method_use* must satisfy the relevant bounds.

- No type instantiation should be given when accessing a static member of a class in a **ldsfld**, **stsfld**, or **call** instruction.

- A type instantiation must be given when accessing a non-static member of a class in a **ldfld**, **stfld**, **call** or **callvirt** instruction.

- Types (a) locating methods; (b) locating fields; (c) specifying extended classes; (d) specifying inherited interfaces; or (e) specifying implemented interfaces may not be type variables.

- All type variables "required at runtime" must be "available at runtime" for all instructions in a method.

- Any type variables used to specify

  (a) a return type or an argument type of a method with code or

  (b) a local type within a method implementation or

  (c) as a part of any instruction within a method implementation;

  must have a bound of at least Object.

- main() and all class constructors must be non-polymorphic

A.1.7 Runtime Type Checks

A RTD (runtime type descriptor) is a token specifying a ground instance of a type, i.e. a type containing no type variables. A runtime check between two type descriptors *td1* and *td2* passes precisely if *td1* is a subtype of *td2* by the above rules.