Imperial College London

Interim Report

Value dependent types for the CLI

Author:
Fraser Waters
fraser.waters08@imperial.ac.uk

Supervisor:
Professor Sophia Drossopoulou

June 13, 2013

Contents

1 Introduction **1**
- 1.1 Value dependent types . . . 1
- 1.2 Motivation . . . 1
  - 1.2.1 Performance . . . 2
- 1.3 The Common Language Infrastructure . . . 3
- 1.4 Project . . . 3

2 Background **5**
- 2.1 The CLI . . . 5
  - 2.1.1 Common intermediate language . . . 5
  - 2.1.2 CLI features . . . 6
    - 2.1.2.1 The type system . . . 6
    - 2.1.2.2 The virtual execution system . . . 7
    - 2.1.2.3 Metadata . . . 8
  - 2.1.3 CLI type system . . . 8
    - 2.1.3.1 Value and Reference types . . . 8
  - 2.1.4 Literal and initonly . . . 8
  - 2.1.5 Properties . . . 9
  - 2.1.6 Generics . . . 10
- 2.2 C++ templates . . . 12
- 2.3 F# units of measure . . . 13
- 2.4 Path dependent types . . . 15
- 2.5 Virtual types . . . 17
- 2.6 First class types . . . 17
- 2.7 Generalized algebraic data types . . . 18
- 2.8 Conclusion . . . 19

3 Type equality constraints **21**
- 3.1 Generalized algebraic data types . . . 21
  - 3.1.1 Example . . . 21
- 3.2 Specification changes . . . 25
  - 3.2.1 Generic constraints . . . 25
  - 3.2.2 Assignment compatibility . . . 25
    - 3.2.2.1 *equal-to* . . . 26
  - 3.2.3 Method calls . . . 26
  - 3.2.4 Metadata tables . . . 26
- 3.3 Implementation changes . . . 27
  - 3.3.1 Runtime . . . 27
  - 3.3.2 corlib . . . 27
  - 3.3.3 Cecil and IKVM . . . 27

3.4 Testing 27
3.5 Testing without a full implementation 29

4 Values for type parameters **30**
4.0.1 Concept of equality 30
4.0.2 Immutability 31
4.0.3 Operations 31

5 Evaluation **32**

6 Conclusion **33**
6.1 Future work 33

Abstract

Value dependent types are a powerful extension to type systems allowing types to be parametrized by terms. This project looks into how value dependent types could be introduced to the CLI, the underlying virtual machine specification for C#, Visual Basic, F# and many other languages, to allow more programs to be succinctly expressed at the CLI level and exposed to these languages.

Chapter 1

Introduction

1.1 Value dependent types

Dependent types allow expression to be statically typed based on values; rather like how parametric types can be based on other types. There are some functional languages such as Agda[1], Coq[4], Idris[6] and Cayanne[11] that support dependent types, but in object oriented languages dependent types are not so common. While fully general value dependent types are rare, some weaker versions, including path dependent types (section 2.4) and virtual types (section 2.5), are used in some mainstream languages. Notably Scala[8] supports both path dependence and virtual types, F#[5] supports units of measure (section 2.3) allowing numbers to be typed based on a unit value, and C++ has templates (section 2.2) that can be parametrized on values.

1.2 Motivation

One of the main motivations for looking into value dependence is for work in graphics and physics applications where most vectors and matrices in the problem domain are small (3 or 4 elements, but no reason this couldn't scale to many more for other applications). Using types such as Vector3, Vector4, Matrix3x4 which are 3 and 4 element float vectors and a 3 by 4 float matrix respectively, make writing code much easier than working with multiple float variables. Currently there is no nice way to represent all the different sizes for types like this in C# (or any other CLI language). Consequently it lead me to the creation of a numeric type generator, a separate program that outputs the source code for a pre defined set of configurations (currently Vector2 to Vector8 and Matrix2x2 up to Matrix4x4). While the use of these types is mostly acceptable extending them is difficult. As shown below using the generator requires writing the code in literal strings, these literals can not be checked at compile time for obvious mistakes and the IDE does not offer auto completion when writing them, the generator has to be run then the emitted code must be compiled to see any problems. The following shows a section of code used to generate all the required dot product functions. `Components` is a list of component indices 0, 1, 2 etc. `WriteLine` writes a string to the code file using the current indent level, `Indent` and `Dedent` increase and decrease the indent level.

```
if (!Type.IsCLSCompliant) { WriteLine("[CLSCompliant(false)]"); }
WriteLine("public static float Dot({0} left, {0} right)", Name);
WriteLine("{");
Indent();
var dotproduct = string.Join("+", Components.Select(
    component => string.Format("left[{0}]*right[{0}]", component)));
WriteLine("return {0};", dotproduct);
Dedent();
WriteLine("}");
```

With value dependence we could write the code directly to be compiled, skipping the generator step, and allowing the use of auto complete and faster iteration times.

```
public static float Dot<int n>(Vector<n> left, Vector<n> right)
{
    float dot = 0;
    for(int i=0; i<n; ++i)
    {
        dot += left[i]*right[i];
    }
    return dot;
}
```

The usage of these types would remain nearly the same, the following shows how they look at the moment compared to what they might look like with value dependence.

Listing 1.1: Current method

```
var a = new Vector3(1, 1, 1);
var b = new Vector3(2, 2, 2);

var ab = b - a;
var dot = Vector.Dot(ab, a);
```

Listing 1.2: Proposal

```
var a = new Vector<3>(1, 1, 1);
var b = new Vector<3>(2, 2, 2);

var ab = b - a;
var dot = Vector.Dot(ab, a);
```

As you can see there isn't a big difference. Users get more benefit with the latter as they can write dependent functions that work for any vector size, as opposed to having to use multiple functions for different sizes.

1.2.1 Performance

There are ways to avoid the generator step now, however they do not have acceptable performance and layout properties to be used.

The simplest way is to define Vector as dynamicly sized (like arrays) this loses all static type safety but does mean functions would only have to be written once, saving us the effort of creating and maintaining the generator but a cost. Vector is no longer purely a value type as it will have a reference field in it, this changes the semantics from the current vector types (due to no user defined copy constructors or assignment operators in the CLI) and also makes them more expensive as they are now tracked by the garbage collector. It also makes inter operating with native APIs such as OpenGL harder as the Vectors will have to be marshaled to correctly copy the elements of their internal array to the native API, the current vector types can just be pinned and pointer copied.

Another way is to define an interface `Vector` that defines the indexing operator and length property, and then write functions using this interface. However we still need to create concrete types for each vector size which requires the generator.

```
interface Vector
{
    int Length { get; }
    float this[int index] { get; }
}
public static float Dot<T>(T a, T b) where T : Vector
{
    float dot = 0;
    for(int i=0; i<a.Length; ++i)
    {
        dot += a[i]*b[i];
    }
    return dot;
}
```

The issue with this interface approach (and this give some suggestion as to how we would want to implement dependent types) is that the loop cannot be unrolled. For high performance code that theses small vector types are supposed to be used for that's an unacceptable trade off, especially as we still have to maintain the generator anyway. With dependent types we could have better performance characteristics and with the preferred flat data layout.

While these vector types are the main motivator for value dependence there are more uses for value dependent types, we explore these in the background section.

1.3 The CLI

The CLI[1] is a specification for a virtual execution environment, that is implemented by Microsoft's CLR[2] (often confused with the .NET branding) and the open source Mono project. It is targeted by VB, C#, F#, IronPython and other languages. It retains a high level of type information, more so than the JVM [3] (which for example has no concept of generic types despite Java supporting them[7]).

As a C# and F# programmer the CLI is a more attractive specification to work with. The ability to retain high level type information allows easy interoperability between separate CLI modules, even with modules compiled using different languages.

However this interoperability starts to fall apart when languages add typing extensions that aren't supported by the CLI. Units of measure in F#, for example, are erased at compile time; therefore other modules which consume an F# module where units of measure were used cannot see, and be type-checked according to, the units. This loss of typing information is not ideal, as it reduces interoperability, and so prompts us to consider adding value dependence as a CLI feature and not just an extension to a current CLI language such as C# or F#. If units could be written in terms of dependent types then we can *fix* them, else at least our extension will not suffer the same problem of interoperability.

Moreover any new features added to the CLI should be backwards compatible and efficient, we need to keep in mind the size of the new types and their instances, the size of the byte code and the speed to process it and the speed and size of the JITed code.

1.4 Project

This project will investigate value dependent types in the CLI. It will be split into 3 parts.

[1]Common Language Infrastructure
[2]Common Language Runtime
[3]Java Virtual Machine

1. To investigate the use and benefits of value dependent typing.
2. To show how value dependent types could be added to the CLI, preferably in a clean and backwards compatible way.
3. If part 2 is successful to implement value dependent typing in Mono.
4. If part 2 is unsuccessful then an through explanation of why it can't be done should be written.

The rest of this report looks more in depth at the CLI and then covers various type system enhancements related to value dependence and value dependence itself. It finishes with an plan for the rest of the project and it's evaluation.

Chapter 2

Background

2.1 The CLI

2.1.1 Common intermediate language

To allow the reader to more easily follow later discussions we will first briefly go over the CLI and CIL. For those more familiar with Java; the CLI can be compared to the JVM, and CIL to Java Bytecode. The CLI runs Common Intermediate Language (CIL) byte code. CIL is a type rich, stack based assembly language. Below is *Hello World* in CIL, it shows the loading of a literal string and then calling a static method. Where possible I have elided instruction offsets in CIL code.

Listing 2.1: Hello world

```
.assembly Hello {}
.assembly extern mscorlib {}
.method static void Main()
{
    .entrypoint
    .maxstack 1
    ldstr "Hello,␣world!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

CIL supports many features not common for low level assembly code, as well as basic operations such as add, jump, load, store. Operations such as field access, method call, object creation, casting etc all have dedicated CIL instructions. A more concrete example of CIL code is 2.3. LoadPpm is a method from a real C# application, it shows the use of locals, calling static and instance methods, method arguments, constructors and properties.

Listing 2.2: LoadPpm C#

```
private static Image LoadPpm(string name)
{
    var path = System.IO.Path.Combine(Toplevel(), "Memorial", name);
    var ppm = new Ibasa.Media.Visual.Pnm(path);

    return new Ibasa.SharpIL.Image(ppm.Image, 0, 0);
}
```

Listing 2.3: LoadPpm CIL

```
.method private hidebysig static
    class [Ibasa.SharpIL]Ibasa.SharpIL.Image LoadPpm(string name)
    cil managed
{
    .maxstack 4
    .locals init (
    [0] string path,
    [1] class [Ibasa.Media]Ibasa.Media.Visual.Pnm ppm)

    L_0000: call string Graphics.Program::Toplevel()
    L_0005: ldstr "Memorial"
    L_000a: ldarg.0
    L_000b: call string
        [mscorlib]System.IO.Path::Combine(string, string, string)
    L_0010: stloc.0
    L_0011: ldloc.0
    L_0012: newobj instance void
        [Ibasa.Media]Ibasa.Media.Visual.Pnm::.ctor(string)
    L_0017: stloc.1
    L_0018: ldloc.1
    L_0019: callvirt instance class [Ibasa.SharpIL]Ibasa.SharpIL.Resource
        [Ibasa.Media]Ibasa.Media.Visual.Pnm::get_Image()
    L_001e: ldc.i4.0
    L_001f: ldc.i4.0
    L_0020: newobj instance void [Ibasa.SharpIL]Ibasa.SharpIL.Image::.ctor(
        class [Ibasa.SharpIL]Ibasa.SharpIL.Resource, int32, int32)
    L_0025: ret
}
```

Therefore, while CIL is targeted by a variety of languages Visual Basic and C# match its semantics most closely. We will use C# code instead of raw CIL when possible in examples. Some CLI/C# features are uncommon in other languages so we we'll briefly go over them. Some of these features may affect ideas for our extensions, others just help simplify explanations and code examples.

2.1.2 CLI features

The CLI supports a number of features to provide a comprehensive target for a variety of languages. The specification defines a type system, a virtual execution system that provides an environment for executing code, and finally metadata that provides a structured way to represent all the information the CLI needs to load types, layout objects, resolve methods, translate CIL, enforce security and set up runtime context boundaries.

The CLI specification is made of three main parts, the type system, the execution system, and metadata. The type system is covered in the next section.

2.1.2.1 The type system

The type system of the CLI is covered in more detail later, in 2.1.3 but briefly covered here as well.

Types in the CLI fall into two categories, value types and reference types. Value types are built in types such as integers and floats, or user defined enumerations and structures. Reference types are the built-in types (`Object` and `String`), objects, interfaces, arrays, delegates, boxed value types, managed and unmanaged pointers.

Value types describe the sequence of bits that make up a value of that type. Reference types describes the sequence of bits that make up the value but also describe the location of that value.

> A type fully describes a value if it unambiguously defines the value's representation and the operations defined on that value.
>
> For a value type, defining the representation entails describing the sequence of bits that make up the value's representation. For a reference type, defining the representation entails describing the location and the sequence of bits that make up the value's representation.

While every value has an exact type it is not always possible to determine the type by looking at the representation of the value. For example the 32 bits used to represent an integer or a float are indistinguishable. Some types it is possible to determine the exact type from the value. Objects are self typing as they explicitly store their type as part of their value.

Some types do not fully describe any values (abstract classes and interfaces). While every value has a type it is not possible to infer the type given the value if the value is of a value type, if the value is a reference type it is always possible to infer the type.

2.1.2.2 The virtual execution system

The VES[1] provides an environment for executing CIL managed code. Managed code is code that can only be run under the *management* of a virtual machine (while originally coined to refer to CIL code only, this terminology has also spread to other *managed* languages such as Java). Unmanaged code in contrary is raw machine code run directly on the processor. This distinction is clearly has some grey area when you consider operation systems, memory permissions and new CPU based code security methods but it's clear enough for our purpose of separating native from managed code.

It supports some built-in data types, control flow constructs, an exception handling model and support for the CIL instruction set.

The supported data types are:

- Signed and unsigned integers of 8, 16, 32 and 64 bits
- 32 and 64 bit IEEE floating point numbers
- Native size signed integer
- Native size unsigned integer, also used as unmanaged pointer
- Native size floating point number (internal, not externally visible)
- Native size object reference to managed memory
- Native size managed pointer

All signed integers are implemented with two's-complement arithmetic.

Managed memory supports garbage collection, as such allocated objects do not not need to be manually deallocated. The locations pointed to by managed pointers and references can be changed during a garbage collection cycle as objects are moved in memory. While a managed pointer or reference is still valid (i.e. reachable) the object will remain in memory.

Exceptions objects are any valid class type (a boxed value type or any class), but not native integers, when an exception is thrown the runtime searches for the first `catch` handler up the stack that covers the throw instruction. If no `catch` handler is found execution is aborted. If a match is found the stack is wound back to that `catch` handler executing all `finally` and `fault` handlers found on the way up.

[1] Virtual Execution System

2.1.2.3 Metadata

Executables and libraries are stored in assemblies. Each assembly is made up of one or more modules. Each module is made up of type and method declarations. This information is stored in a structured way and is known as metadata. The CLI uses metadata to locate and load classes, lay instances out in memory, resolve method invocations, translate CIL to native code, enforce security and set up runtime context boundaries.

2.1.3 CLI type system

2.1.3.1 Value and Reference types

The CLI (and C#) differentiates between value types (structs) and reference types (classes). Value types are allocated inline, either on the stack or as part of a containing types allocation. Reference types are allocated on the heap and referred to by a pointer (called a reference); these are tracked by the garbage collector. To compare this to C++, `Foo` is a value type, while `Foo*` is a reference type, the semantics are similar.

2.1.4 Literal and initonly

Fields in the CLI can be marked as `initonly`, and if they are static fields, `literal`. Properties do not support either of these modifiers.

A `static literal` field has no space allocated for it in the metadata, instead any reference to that field must have the literal value copied into the use site, as such literal fields must be a primitive type (int, float, string, etc) In C# the keyword `const` is used instead of `literal`.

An `initonly` field can only be written to by a constructor method (or if static by the type initializer method). Other methods can only load from the field. In C# the keyword `readonly` is used instead of `initonly`. The property initonly is not transitive, for example the following C# mutates a readonly Pair field and is valid code.

```
class Pair
{
    public int A;
    public int B;

    public Pair(int a, int b)
    {
        A = a;
        B = b;
    }
}

class Program
{
    public readonly Pair MyPair;

    public Program()
    {
        MyPair = new Pair(1, 2);
    }

    static Main()
    {
        var program = new Program();
        //program.MyPair = new Pair(3, 4); not valid
        program.MyPair.A = 3; // valid
    }
}
```

This is a very weak concept of immutability, and when adding user defined types to value dependence could present problems.

2.1.5 Properties

As pointed out above the CIL has instructions for field access but it also has first class support for properties. Properties support get and set methods (both optional), which do not have to have the same visibility (it's valid to have a public get and private set). Properties can also have parameters which turns them into indexers. In the CIL code these look similar to method calls but in C# they look like field access.

```
class Square {
    public int Length;
    public int Area { get { return Length * Length; } }

    static void Main() {
        Square sq = new Square();
        sq.Length = 4;
        Console.WriteLine(sq.Area); // outputs 16
    }
}
```

The corresponding CIL follows, note the method call for `get_Area` on line 16 in `Main`. While the property getter is just a method it is marked up specially in the `.property` clause so that other tools can treat it specially.

```
1   .class private auto ansi beforefieldinit Square
2   extends [mscorlib]System.Object
3   {
4       .method private hidebysig static void Main() cil managed
5       {
6           .entrypoint
7           .maxstack 2
8           .locals init ([0] class Square sq)
9
10          newobj instance void Square::.ctor()
11          stloc.0
12          ldloc.0
13          ldc.i4.4
14          stfld int32 Square::Length
15          ldloc.0
16          callvirt instance int32 Square::get_Area()
17          call void [mscorlib]System.Console::WriteLine(int32)
18          ret
19      }
20
21      .property instance int32 Area
```

```
    {
        .get instance int32 Square::get_Area()
    }

    .field public int32 Length

    .method public hidebysig specialname instance int32 get_Area() cil managed
    {
        .maxstack 2

        ldarg.0
        ldfld int32 Square::Length
        ldarg.0
        ldfld int32 Square::Length
        mul
        ret
    }
}
```

Adding parameters to a property changes it into an indexer. The CIL remains pretty similar but C# treats indexers very differently. C# only allows indexers called `this`, `this` is translated to `Item` when compiled to CIL. If a type has an indexer `Item` then values of the type can have an indexer expression appended to them (like accessing an array).

```
class StringIntMap {
    public int this[string key] {
        get { ...; } set { ...; } // assuming a sensible implementation
}

void Main() {
    StringIntMap map = new StringIntMap();
    map["test"] = 1;
    Console.WriteLine(map["test"]); // outputs 1
}
```

Other languages (such as F# and C++/CLI) allow named indexers. While they both treat `Item` specially in the same way as C# they allow parameters on explicitly named properties as well.

```
let row = 1 in matrix.rows[row]
```

2.1.6 Generics

The CLI supports parametric polymorphic types via generics types. Generic types are parametrized on other types (value dependence would allow types to also be parametrized on values). The MSR White paper [17] describes some initial design considerations to do parametric polymorphism in COM+ (the original name for what became the CLI and .NET). While the final design and implementation that shipped with .NET differs slightly from the design presented in [17], the paper does give an insight into what we need to be thinking about while designing value parametrics. One major change between the proposal and the final implementation is the use of value types as generic arguments. It was originally thought that allowing value types as generic arguments was not worth the performance cost of having to re-jit all code that used them; the final implementation decided that the expressivity and performance increase from not boxing out weighed this downside. It's worth taking some time to look at how generics ended up being specified in ECMA-335[10] and implemented in Mono (due to copyright reasons we can't look at Microsoft's open source CLR code).

Generics are defined in section II.9 of [10]. A type in the CLI can have a fixed generic arity (that is generics are not variadic), the parameters are unnamed and are accessed by index (either !0 or for type parameters and !!0 for method parameters). Each type parameter may be constrained by a number of properties, including constraints on being a value or reference type, having a defined base class or interface or being default constructable. Type parameters can be value or reference types; this is a marked difference from the suggestion in [17] which suggested that value types should not be allowed due to having to re-JIT the types code for each value type.

Generics allow the CLI to represent types such as `List<T>` while retaining run time information such that the run time type of `List<object>` is different to `List<int>`[2]. `List<int>` is also special in that `int` is a value type and yet the run time can use a `List<int>`without causing excessive boxing of values.

If we look at the definition of `List<T>` in Microsoft's distribution of .NET 4.0 we can see how the generic parameter is declared and used.

```
1  .class public auto ansi serializable beforefieldinit List'1<T>
2  extends [mscorlib]System.Object
3  implements System.Collections.Generic.IList'1<!0>,
4      System.Collections.Generic.ICollection'1<!0>,
5      System.Collections.Generic.IEnumerable'1<!0>,
6      System.Collections.IList,
7      System.Collections.ICollection,
8      System.Collections.IEnumerable
9  { ... }
```

The declaration `.class public auto ansi serializable beforefieldinit List‘1<T>` declares a new class type with one generic parameter `T`, which has no constraints. The `implements` clause lists interfaces implemented by `List‘1<T>`, the first three of these interfaces are themselves generic. On line 3 the `System.Collections.Generic.IList‘1` syntax indicates that we mean the generic `IList` with one parameter `‘1`, while <!0> refers to the first generic class parameter `T`, and passes that as the type argument to `IList`.

Generic parameters can also be constrained, a run length compressed list for example would require that the type it stored had an equality operator. The `IEquatable<T>` interface defines a method `bool Equal(T value)`, so if a type `T` inherits from `IEquatable<T>` then it can be compared equal to other values of its type. Adding the constraint that the first generic parameters has this property is shown here. Note the `(IEquatable‘1<!0>)` before the `T`.

```
.class public auto ansi sealed beforefieldinit
    CompressedList'1<(IEquatable'1<!0>) T>
extends [mscorlib]System.Object
implements System.Collections.Generic.IEnumerable'1<!0>,
    System.Collections.IEnumerable
{ ... }
```

Sometimes it is necessary to distinguish between instantiated and non-instantiated generic types. Common terminology for this, as used in ECMA-335, is close and open generic types. A closed generic type is one that has no unbound type parameters, conversly an open generic type is a generic type that has at least one unbound type paramter.

Listing 2.4: Open and closed type in C# syntax

```
Dictionary<TKey, TValue> a; // open
```

[2]In contrast theses types would be equivalent in the JVM.

```
Dictionary<TKey, int> b; // open
Dictionary<string, int> c; // close
```

2.2 C++ templates

C++ templates allow functions and types to be parametrized by types or values. Templates are a turning complete language by themselves making them very general, however most implementations of templates simply perform substitution at compile time leading to a large amount of generated code that then has to be reduced by looking for similarities (in contrast to CLR and Mono generics that duplicate very little code), and with large use cases substantial slowdowns to compilation time.

Many libraries including the standard library make use of templates, particularity the parametrization on types. Parametrization on values is less used but it's similarities to value dependence make it worth looking at, to this end we will look at a few examples from the standard library, Boost[2] and CML[3].

The standard C++ library uses value templates in a few places including `std::ratio` and the random number generation library. `std::ratio` is a compile time rational number added in C++11, it reduces the numerator and denominator to lowest terms at compile time. The random number generator uses template values to set generator parameters such as the constants `a`, `c` and `m` to be used in `std::linear_congruential_engine`.

```
template<
    class UIntType,
    UIntType a,
    UIntType c,
    UIntType m
> class linear_congruential_engine;
```

The open source Boost[2] libraries make use of value templates much more, using them in obvious ways in the Array library, which is for safer arrays using a new class `Array<typename T, int N>`, but also scattered throughout the other libraries. For example in `Spirit::Qi`, a parser combinator library, the type `unit_parser` is templated on the type name of the integer type to return but also on the values of the radix and minimum and maximum digits to parse.

Finally CML[3] uses value templates to define the sizes of vectors and matrices, this is similar to our motivating example in C#. Vector and matrix are templated on two types `ElementT` and `StorageT`. `ElementT` is the element type, float, double, int or another type that supports the same operations. `StorageT` is a type that provides access to the elements, either by pointing to an external data source or storing the data itself. Two of the built in storage types (fixed and external) are templated on the value of how many elements they store/point to. When using these statically sized storage types you get extra static type safety that you're not mixing vector sizes in operations.

```
cml::vector<float, fixed<3>> a(1,0,0);
cml::vector<float, fixed<2>> b = a; // compile error
cml::matrix<float, fixed<2,2>> i(1, 2, 3, 4);
cml::matrix<float, fixed<3,3>> j = i; // compile error
```

It's worth noting that although this example looks like the constructors match the value passed to fixed they are actually pre declared constructors for all normal sizes, using the wrong constructor will either leave some elements uninitialized or not use some of the values passed in. Real variadic parameters that matched the dimension of the vector/matrix would be better, and with the new features of C++11 might be possible.

2.3 F# units of measure

F# has the ability to markup number values with units of measure that allow checking of units at compile time. This extra checking can prevent mistakes such as that which brought down the Mars Climate Orbiter in 1999. The Orbiter crashed because of a mismatch between Imperial and Metric units in force calculation. A very expensive mistake as the mission cost $327.6 million[18].

Units of measure are declared as opaque types marked up with the `Measure` attribute.

```
[<Measure>] type meter
```

They can also be declared as equal to other units, for example milliliters as cubic centimeters.

```
[<Measure>] type ml = cm^3
```

The normal unit operators such as multiplication, division and powers are usable and can be worked out by the type inference engine. For example in the following, code type inference correctly identifies `distance` as type `float<meter>`.

```
let speed = 55.0<meter/second>
let time = 3.5<second>
let distance = speed * time;

speed     : float<meter/second>
time      : float<second>
distance  : float<meter>
```

The compiler will normalize units of measure to a standard form, from the MSDN documentation[14]

> "Unit formulas that mean the same thing can be written in various equivalent ways. Therefore, the compiler converts unit formulas into a consistent form, which converts negative powers to reciprocals, groups units into a single numerator and a denominator, and alphabetizes the units in the numerator and denominator."

Units of measure are a common praise of F# and provided a valuable case study for us to use in our type system extension.

F# units of measure are checked at compile time, implemented as a sort separate from the standard types. However all units units information is erased from the run time. Therefore values cast to `Object` cannot be recast to a measured type safely at run time, but also these units cannot be exposed as part of a public interface to be consumed by other CLI languages such as C# or VB.

While they are implemented as a separate sort they behave somewhat like values of a standard type (with operations for multiplication and division). A system that allowed them to be values of a Measure type (rather than a separate sort) while retaining the current features (including inference) would be impressive and something our system should strive for.

Listing 2.5: Example Unit type

```
public sealed class Unit
{
    public static Unit One
    {
        get
        {
            return new Unit(new List<Tuple<string, int>>());
        }
    }
```

```
private readonly List<Tuple<string, int>> Units;

private Unit(List<Tuple<string, int>> units)
{
    Units = units;
}

public Unit(string unit)
{
    Units = new List<Tuple<string, int>>();
    Units.Add(Tuple.Create(unit, 1));
}

public static Unit operator *(Unit a, Unit b)
{
    return new Unit(Product(a.Units, b.Units));
}

public static Unit operator /(Unit a, Unit b)
{
    return new Unit(Product(a.Units, Reciprocal(b.Units)));
}

public static Unit operator ^(Unit a, int power)
{
    if (power == 0)
        return One;
    return new Unit(Power(a.Units, power));
}

private static List<Tuple<string, int>> Normalize(
    List<Tuple<string, int>> units)
{
    var groups = units.GroupBy(tuple => tuple.Item1);
    var sums = groups.Select(group =>
        Tuple.Create(group.Key, group.Sum(unit => unit.Item2)));
    var filter = sums.Where(unit => unit.Item2 != 0);
    var sorted = filter.OrderBy(unit => unit.Item1);
    return sorted.ToList();
}

private static List<Tuple<string, int>> Product(
    List<Tuple<string, int>> a, List<Tuple<string, int>> b)
{
    return Normalize(a.Concat(b).ToList());
}

private static List<Tuple<string, int>> Power(
    List<Tuple<string, int>> a, int power)
{
        return a.Select(unit =>
            Tuple.Create(unit.Item1, unit.Item2 * power)).ToList();
    }

    private static List<Tuple<string, int>> Reciprocal(
        List<Tuple<string, int>> units)
    {
        return units.Select(unit =>
            Tuple.Create(unit.Item1, -unit.Item2)).ToList();
    }

    public override bool Equals(object obj)
    {
        if (obj is Unit)
        {
            var other = obj as Unit;

            return
                Units.Count == other.Units.Count &&
                Units.Zip(other.Units, (a, b) =>
                    a.Item1 == b.Item1 && a.Item2 == b.Item2).All(b => b);
        }
        return false;
    }

    public override string ToString()
    {
        return string.Join(" ", Units.Select(unit =>
            string.Format("{0}^{1}", unit.Item1, unit.Item2)));
    }
}

public static void Example()
{
    Unit meters = new Unit("m");
    Unit seconds = new Unit("s");
    Unit metersPerSecond = meters / seconds;
}
```

2.4 Path dependent types

Path dependent types like those found in Scala are similar to value dependent types in that they depend on the value of the object that created them, but they are not as general. An example of path dependence in Scala is the following Board and Coordinate example[9].

```
case class Board(length: Int, height: Int)
{
    case class Coordinate(x: Int, y: Int)
    {
        require(0 <= x && x < length && 0 <= y && y < height)
    }
    val occupied = scala.collection.mutable.Set[Coordinate]()
}

val b1 = Board(20, 20)
val b2 = Board(30, 30)
var b3 = b1
val c1 = b1.Coordinate(15, 15)
val c2 = b2.Coordinate(25, 25)
b1.occupied += c1
b2.occupied += c2
b3.occupied += c1
// Next line doesn't compile
b1.occupied += c2
```

Here the type of `c1` and `c2` depend on the values `b1` and `b2`. Not that it is in fact the values not these specific identifiers that are the dependence, as shown on line 17. Path dependence in the type system does not allow line 19, which is stricter than just inner classes in Java.

Path dependence is an extension of the fact that in Scala and Java inner classes are created via an instance of the outer class and maintain a reference to their creator. I call the creation via an instance of the outer class an instance inner types, as opposed to static inner types that do not require an instance of the outer class. The CLI does not support path dependent types or instance inner types, the only difference between inner and outer class in the CLI is viability (that is an inner class can be made private and thus only be accessed by the outer class). While it's possible to require a reference to the outer class as part of the inner class's constructor it is not a requirement. While instance created inner classes and then path dependence could be added at the language level this leads to the risk that Scala ran into where the virtual machine reflection system no longer resembled the language type system, thus pushing for the implementation of a whole new reflection system to be built.

Therefore if we are to investigate the addition of adding path dependent types we also need to add instance inner types to the CLI. Alternatively we could try to design value dependence such that the following was possible.

```
class Board
{
    int length, height;

    public Board(int length, int height)
    {
        this.length = length;
        this.height = height;
    }

    class Coordianate<Board b>
    {
        public Coordinate(int x, int y)
        {
            require(0 <= x && x < b.length && 0 <= y && y < b.height)
        }
    }

    Set<Coordianate<this>> occupied = new Set<Coordinate<this>>;
}
```

Allowing the value parameter to be any type is much more general than path dependence, In this case `Coordinate` would not even need to be an inner class of `Board`. However this is a very ambitious addition and if it's even possible is uncertain.

2.5 Virtual types

Virtual types are also found in Scala, they allow a subclass to override a type variable in the super class. In the following example the type `T` declared in class `A` is made more specific in the subclass `B`.

```
class A
{
    type T
    abstract T foo();
}

class B
{
    override type T = String
    override T foo() { return "string"; }
}
```

While virtual types can be useful everything they accomplish can also be done with generics, albeit with sometime much more syntax. [12] shows how the same program can be expressed with virtual types or parametrized types. While one way is often more elegant than the other you gain little in supporting both. As parametrized types are already supported by the CLI virtual types are not hugely interesting.

2.6 First class types

Cayenne[11] is a language with support for dependent types and first class types (i.e. types can be be used like values). As Cayenne is a functional language inspired by Haskell, it's unlikely we can lift ideas straight from it to be used in the CLI, however it provides an example of a very general dependent types system. Two core features of Cayenne are dependent functions and dependent records. Dependent functions allow a function return type to depend on the value of the parameter, as shown in the following example from [11].

```
printfType :: String -> #
PrintfType "" = String
PrintfType ('%':'d':cs) = Int    -> PrintfType cs
PrintfType ('%':'s':cs) = Stirng -> PrintfType cs
PrintfType ('%':_:cs)   =           PrintfType cs
PrintfType (_:cs)       =           PrintfType cs

printf :: (fmt::String) -> PrintfType fmt
printf fmt = pr fmt ""

pr :: (fmt::String) -> String -> PrintfType fmt
pr "" res = res
pr ('%':'d':cs) res = \(i::Int) -> pr cs (res ++ show i)
pr ('%':'s':cs) res = \(s::String) -> pr cs (res ++ s)
pr ('%':c:cs) res   = pr cs (res ++ [c])
pr (c:cs) res       = pr css (res ++ [c])
```

In this example the type of `printf` depends on the value of the parameter fmt. This also shows how types and values are treated equally in Cayenne. The type `#` is the type of all types (normal notation is `*` but `#` was chosen to avoid clashes with the infix operator `*`).

2.7 Generalized algebraic data types

Generalized algebraic data types (GADTs) are predominately a feature of functional languages. They are an extension to algebraic data types. They allow more expression in data type constructors, in particular they allow pattern matching and more general recursion in a data constructor. The common example is a type for terms in a small language; with GADTs it's possible to express constraints on the expression trees that are not expresable in normal ADTs.

Listing 2.6: GADT

```
data Exp t where
    Lit :: Int -> Exp Int
    Plus :: Exp Int -> Exp Int -> Exp Int
    Equals :: Exp Int -> Exp Int -> Exp Bool
    Cond :: Exp Bool -> Exp a -> Exp a -> Exp a
```

This data type will only allow correct instantations of `Exp` as paramters. The constraint that `Plus` takes two `Int` expressions and returns a new Int expression is expressed. As is the constraint that `Cond` must take a `Bool` expression and two other expression of the same type returning an expression of that type. Without GADTs these constraints cannot be expressed, the following shows the same type as an ADT.

Listing 2.7: ADT

```
data Exp
    = Lit Int
    | Plus Exp Exp
    | Equals Exp Exp
    | Cond Exp Exp Exp
```

The first expression passed to `Cond` must evaluate to a boolean result (from `Equals`), but the type system cannot express that. The following expression tree is valid with the ADT type, and invalid with GADTs.

```
Cond (Lit 1) (Lit 2) (Equals (Lit 3) (Lit 4))
```

There are more examples that can be statically checked with GADTs such as lists that have their size as part of their type and statically typed printf functions.

The use of GADTs in object orientated languages is less common than in functional languages (Haskell has supported GADTs for over 10 years) but [13] shows how GADT programs can be expressed in C# with some modifications to the language. The two modifications proposed by [13] are an extension of generic constraints and an extension of the switch statement.

The extension to generic constraints would allow equality constraints on generic types, section 3.1 (Equational constraints for C#) of [13] describes this extension. This would allow a generic type to be declared equal to another type, this would be checked statically at compile time.

We'll use a different example from the expressions code above, as the expression type is much larger expressed in C#. Instead we'll look at list flatten methods. A list flatten method could check that the list was a list of lists by the addition of the `where T=List<U>` clause.

Listing 2.8: C# GADT

```
public abstract class List<T> {
    public abstract List<T> Append(List<T> list);
    public abstract List<U> Flatten<U>() where T=List<U>;
}

public class Nil<T> : List<T> {
    public override List<U> Flatten<U>() {
        return new Nil<U>;
    }
}

public class Cons<T> : List<T> {
    T head; List<T> tail;
    public override List<U> Flatten<U>() {
        return this.head.Append(this.tail.Flatten());
    }
}
```

Calling `Flatten` on a `List<T>` would statically check that `T=List<U>` where `U` is any type. Thus in the method body of flatten we can assume that the type of `head` is `List<U>` which has an `Append` method. While [13] suggests this addition of type equality constraints as a C# extension; generic constraints are currently encoded at the CLI level and so we could add this as a CLI extension. Adding type equality constraints at the CLI level would allow it to be added to C# and other languages more easily.

The second proposal is an extension to the switch statement to allow switching on types, binding type `variables` in switch case clauses and matching multiple expressions.

```
switch (e1, e2)
    case (Lit x, Lit y):
        return x.value == y.value;
    case (Tuple<A,B> x, Tuple<C,D> y):
        return Eq(x.fst, y.fst) && Eq(x.snd, y.snd);
    default:
        returna false;
}
```

While standard switch statements are a language feature (at the CLI level they are encoded through a sequence of if statements) the authors point out that support at the CLI level for a match-and-bind primitive would be useful (see the end of section 3.4 in [13]) as their switch extension is currently difficult to translate to CIL code, having to rely on run time reflection and generic methods to bind correctly.

2.8 Conclusion

Having looked at all these type systems we can see that some systems are more powerfull, while other are equal in power but differ in expresivity.

C++ templates being Turing complete are the most powerful system we've looked at, but that comes with it's downsides. Efficiently compiling templates such that the final code is small and fast is difficult, they also make the language unsound as a template can recurse forever (although most compilers have hard limits to this).

The next most general is **value dependence**. Value dependence allowed types to be constructed based on values. While this is powerful allowing arbitrary values as parameters is undecidable, as it amounts to determining whether two different programs produce the same result. In chapter 30.5 of [15] is a warning about dependent types:

> Unfortunately, the power of dependent types is a two-edged sword. Blurring the distinction between checking types and carrying out proofs of arbitrary theorems does not magically make theorem proving simple - on the contrary, it makes type checking computationally intractable! Mathematicians working with mechanical proof assistants do not just type in a theorem, press a button, and sit back to wait for a Yes or No: they spend significant effort writing proof scripts and tactics to guide the tool in constructing and verifying a proof. If we carry the idea of correctness by construction to its limits, programmers should expect to expend similar amounts of effort annotating programs with hints and explanations to guide the type checker. For certain critical programming tasks, this degree of effort may be justified, but for day-to-day programming it is almost certainly too costly.

The CLI as a mainstream day-to-day infrastructure would certainly not benefit from an extension that required significant expenditure of programmer time. As such, we do not actually want to make our system too powerful, we want to find a balance between opening up opportunities for optimization and expressivity and the cost of annotation and understanding.

GADTs come in next. Section 2.7 explored how GADTs can be expressed in C# with some modifications. As generic constraints are already stored in the metadata, taking these ideas and expanding them to cover type equality should be simple.

We've seen how **virtual types** are equivalent to generics. As the CLI already supports generics further investigation of virtual types seems unnecessary. Finally, path dependence is a simpler case of value dependence as are F# units of measure.

So while full value dependence may be too much, GADTs aren't enough leading us to think about an extension somewhere in between the two.

$$\begin{gathered} \textit{Templates} \\ \textit{Value dependence} \\ \textit{Our extension?} \\ \textit{GADTs} \sim \textit{Generics} + \textit{Type equality constraints} \\ \textit{Virtual types} \sim \textit{Generics} \end{gathered}$$

The following table shows some of the differances and similarites between these systems. The Type safe printf column indicates if the system can express a printf like function in a type safe way. The type sized lists column indicates if the system can express lists that carry their size as part of their type.

	Turing complete	Type safe printf	Type sized lists	Decidable	Units of measure	Path dependence
Templates	Yes	Yes	Yes	No	Yes	?[3]
Value dependence	No[4]	Yes	Yes	No	Yes	Yes
GADTs	No	Yes	Yes	Yes	No[5]	No
Generics	No	No	No	Yes	No	No

[3]Being Turing complete it feels this should be yes, however we can't find any material to suggest either way

[4]Agda and Coq aren't Turing complete.

[5]Yes if we allow new Kinds[16].

Chapter 3

Type equality constraints

3.1 Generalized algebraic data types

The first extension will be to add type equality constraints to the CLI. Type equality constraints for C# are described in [13], we will be using these ideas but for the CLI not C#. The basic idea is to extend generic constraints to allow equality constraints on generic types, section 3.1 (Equational constraints for C#) of [13] describes this extension. This would allow a generic type to be declared equal to another type, this would be checked statically at compile time. For example a list flatten method could check that the list was a list of lists by the addition of the `where T=List<U>` clause.

```
public abstract class List<T> {
    ...
    public abstract List<T> Append(List<T> list);
    public abstract List<U> Flatten<U>() where T=List<U>;
}

public class Nil<T> : List<T> {
    public override List<U> Flatten<U>() {
        return new Nil<U>;
    }
}

public class Cons<T> : List<T> {
    T head; List<T> tail;
    public override List<U> Flatten<U>() {
        return this.head.Append(this.tail.Flatten());
    }
}
```

Calling `Flatten` on a `List<T>` would statically check that `T=List<U>` where `U` is any type. Thus in the method body of flatten we can assume that the type of `head` is `List<U>` which has an `Append` method. While the paper suggests this as a C# extension generic constraints are currently encoded at the CLI level and so we could add this as a CLI extension, thus allowing this to be added to C# and other languages easily.

3.1.1 Example

The following shows a minimal list example, in both C# and CIL.

Listing 3.1: Type equality constraints in extended C#
Extension of listing 2.8

```
public abstract class List<T>
{
    public abstract List<T> Append(List<T> list);
    public abstract List<U> Flatten<U>() where T=List<U>;
}

public class Nil<T> : List<T>
{
    public override List<T> Append(List<T> list)
    {
        return list;
    }
    public abstract List<U> Flatten<U>() // type constraints are inherited
    {
        return new Nil<U>();
    }
}

public class Cons<T> : List<T>
{
    T Head;
    List<T> Tail;

    public Cons(T head, List<T> tail)
    {
        Head = head;
        Tail = tail;
    }

    public override List<T> Append(List<T> list)
    {
        return new Cons<T>(Head, Tail.Append(list));
    }

    public override List<U> Flatten<U>() // type constraints are inherited
    {
        return Head.Append(Tail.Flatten<U>()); // invalid in standard C#
    }
}
```

Listing 3.2: Corresponding CIL

```
.class public abstract auto ansi beforefieldinit List<T>
extends [mscorlib]System.Object
{
    .method family hidebysig specialname rtspecialname instance void .ctor()
        cil managed
    {
        .maxstack 8
        ldarg.0
        call instance void [mscorlib]System.Object::.ctor()
        ret
    }

    .method public hidebysig newslot abstract virtual instance class
        Test.List`1<!T> Append(class Test.List`1<!T> list) cil managed
    {      }

    .method public hidebysig newslot abstract virtual instance class
        Test.List`1<!!U> Flatten<= T List<!!0> U>() cil managed
    {      }
}

.class public auto ansi beforefieldinit Nil<T>
    extends Test.List`1<!T>
{
    .method public hidebysig specialname rtspecialname instance void .ctor()
        cil managed
    {
        .maxstack 8
        ldarg.0
        call instance void Test.List`1<!T>::.ctor()
        ret
    }

    .method public hidebysig virtual instance class
        Test.List`1<!T> Append(class Test.List`1<!T> list) cil managed
    {
        .maxstack 1
        ldarg.1
        ret
    }

    .method public hidebysig virtual instance class
        Test.List`1<!!U> Flatten<= t list<!!0> U>() cil managed
    {
        .maxstack 1
        newobj instance void Test.Nil`1<!!U>::.ctor()
        ret
    }
}

.class public auto ansi beforefieldinit Cons<T>
extends Test.List`1<!T>
{
    .method public hidebysig specialname rtspecialname instance void
        .ctor(!T head, class Test.List`1<!T> tail) cil managed
    {
        .maxstack 2
        ldarg.0
        call instance void Test.List'1<!T>::.ctor()
        ldarg.0
        ldarg.1
        stfld !0 Test.Cons'1<!T>::Head
        ldarg.0
        ldarg.2
        stfld class Test.List'1<!0> Test.Cons'1<!T>::Tail
        ret
    }

    .method public hidebysig virtual instance class
        Test.List'1<!T> Append(class Test.List'1<!T> list) cil managed
    {
        .maxstack 3
        ldarg.0
        ldfld !0 Test.Cons'1<!T>::Head
        ldarg.0
        ldfld class Test.List'1<!0> Test.Cons'1<!T>::Tail
        ldarg.1
        callvirt instance class Test.List'1<!0>
            Test.List'1<!T>::Append(class Test.List'1<!0>)
        newobj instance void Test.Cons'1<!T>::.ctor(!0, class Test.List'1<!0>)
        ret
    }

    .method public hidebysig virtual instance class
        Test.List'1<!!U> Flatten<= T List<!!0> U>() cil managed
    {
        .maxstack 2
        nop
        ldarg.0
        ldfld !0 Test.Cons'1<!T>::Head
        ldarg.0
        ldfld class Test.List'1<!0> Test.Cons'1<!T>::Tail
        callvirt instance class Test.List'1<!!0>
            Test.List'1<!T>::Flatten<!!U>()

        // the following callvirt would not verify in the standard CLI

        callvirt instance class Test.List'1<!0>
            Test.List'1<!!U>::Append(class Test.List'1<!0>)
        ret
    }

    .field private !T Head
    .field private class Test.List'1<!T> Tail
}
```

The syntax here is purely to demonstrate the intuition of the feature. Exact syntax will be expanded on as we explore how this can be added to the CLI specification.

3.2 Specification changes

3.2.1 Generic constraints

Section II.9.11 (Constraints on generic parameters)[10] specifies generic constraints. A type parameter that has been constrained must be instantiated with an argument that is assignable to each of declared constraints, and that satisfies all special constraints.

The special constraints are, `+`, `-`, `class`, `valuetype` and `.ctor`. `class` constrains the argument to be a reference type. `valuetype` constrains the argument to be a value type, except for any instance of `System.Nullable<T>`. `.ctor` constrains the argument to a type that has a public default constructor (implicitly this means all value types as value types always have a public default constructor). Finally `+` and `-` are used to denote the parameter is covariant or contravariant respectively.

While it might seem that this is a good place to add our extension type equality constraints are a constraint on the entire parameter list, not on each individual parameter. There's also the potential to add an equality constraint to a non-generic method.

```
class Foo<T>
{
    public void Bar(List<int> list) where T = int
    {
        ...
    }

    public void Baz(List<string> list) where T = string
    {
        ...
    }
}
```

In this example `Bar` can only be called if `Foo<T>` was initialized with `int`, and `Baz` only if it was initialized with `string`. A similar thing can be done with non-generic inner types. So we need to look to add this new syntax somewhere separate to the generic parameter list. Preferably it would have similar syntax for both methods and types (as generic parameters look the same on a type declaration or method declaration). A type declaration currently follows the pattern "`.class ClassAttr* Id ['<' GenPars '>'] [extends TypeSpec [ implements TypeSpec] [',' TypeSpec]*]`", while method declarations follow the pattern "`.method MethAttr* [CallConv] Type [marshal '(' [NativeType] ')'] MethodName ['<' GenPars '>'] '(' Parameters ')' ImplAttr*`".

Adding a new clause "`where [Type '=' Type[',' Type '=' Type]*]` to method declarations after the parameter list gives us a list of Types that must be equal to other types. It's not strictly necessary to have this clause on type declarations as for top level types it makes very little sense and for inner types it can be emulated be adding the clause to each method.

3.2.2 Assignment compatibility

Assignment compatibility is defined in section I.8.7 of [10], further to this verification assignment compatibility is defined in III.1.8.1.2.3. Verification assignment compatibility is mostly defined in terms of general assignment compatibility from I.8.7.3.

Verification assignment compatibility is used be the verifier for determining if method calls, field references and loads and stores are valid for a given type and signature. If verification assignment compatibility is extended to understand type equality constraints then operations that were unverifiable but type correct can now be checked as verifiable as well.

Adding another rule to *verifier-assignable-to* to use the rules for equality constraints is all that is needed to enhance this part of the system.

- T is *equal-to* U.

3.2.2.1 *equal-to*

equal-to is used to determine if two type names refer to the same actual type. It uses the both the global typing environment Γ and the equality constraints on the current method ϵ which defines equal types.

$$\text{eq-hyp}\ \frac{T = U \in \epsilon}{\Gamma, \epsilon \vdash T = U}$$

$$\text{eq-con}\ \frac{T = U \in \epsilon \qquad \Gamma \vdash C < T > ok}{\Gamma, \epsilon \vdash C < T >= C < U >}$$

$$\text{eq-decon}\ \frac{\Gamma, \epsilon \vdash C < T >= C < U >}{\Gamma, \epsilon \vdash T = U}$$

$$\text{eq-refl}\ \frac{\Gamma \vdash T\, ok}{\Gamma, \epsilon \vdash T = T}$$

$$\text{eq-sym}\ \frac{U = T \in \epsilon}{\Gamma, \epsilon \vdash T = U}$$

$$\text{eq-sym}\ \frac{\Gamma, \epsilon \vdash T = U \qquad \Gamma, \epsilon \vdash U = V}{\Gamma, \epsilon \vdash T = V}$$

3.2.3 Method calls

Any method calls must be checked that any equality constraints on the method being called can be satisfied by the current context. This is done by checking that for each constraint listed on the method ($T = U$) T and U are *equal-to* in the current context after applying substitution for generic parameters.

$$\text{call}\ \frac{\Gamma \vdash m :< \bar{U} > \ where\, \epsilon' \qquad \forall e \in (\epsilon'[\bar{T}/\bar{U}]).\Gamma, \epsilon \vdash e}{\Gamma, \epsilon \vdash m < \bar{T} >}$$

This is an addition to the verifiability rules for the call, calli and callvirt instructions.

3.2.4 Metadata tables

Metadata tables are specified in section 22 (Metadata logical format: tables). We're mostly interested in section 22.21 (GenericParamConstraint), which explains how generic constraints are stored in the assembly.

> The GenericParamConstraint table has the following columns:
>
> - Owner (an index into the GenericParam table, specifying to which generic parameter this row refers)
> - Constraint (an index into the TypeDef, TypeRef, or TypeSpec tables, specifying from which class this generic parameter is constrained to derive; or which interface this generic parameter is constrained to implement; more precisely, a TypeDefOrRef (§24.2.6) coded index)
>
> The GenericParamConstraint table records the constraints for each generic parameter. Each generic parameter can be constrained to derive from zero or one class. Each generic parameter can be constrained to implement zero or more interfaces.
>
> Conceptually, each row in the GenericParamConstraint table is owned by a row in the GenericParam table.
>
> All rows in the GenericParamConstraint table for a given Owner shall refer to distinct constraints.

We need a similar table for our equality constraints. It will need an owner (a type or method) and then two types that are constrained to be equal within the owner. Another column that isn't necessary but could be useful is a flags field describing the relationship between the two types, currently this would always be set to equal, however the system could be extended at a later date to allow less than and greater than relationships as well.

So the TypeRelationshipConstraint table has the following columns:

- Owner (an index into the MethodDef table, specifying the Method to which this constraint applies)
- Flags (a 1-byte flag bitmask currently always set to 0. To be used for extensions)
- ConstraintA (an index into the TypeDef, TypeRef, or TypeSpec tables, specifying the first type; more precisely, a TypeDefOrRef (§24.2.6) coded index)
- ConstraintB (an index into the TypeDef, TypeRef, or TypeSpec tables, specifying the second type; more precisely, a TypeDefOrRef (§24.2.6) coded index)

3.3 Implementation changes

3.3.1 Runtime

The runtime needs to be changed to load and understand equality constraint metadata, these changes will be focused in mono/metadata/metadata.h/c and mono/metadata/reflection.h/c. The TypeRelationship-Constraint table is very similar to the GenericParamConstraint table, the current code for GenericParam-Constraint can guide the new code for TypeRelationshipConstraint.

Verification is done in verify.c. Additions need to be made to verify that type relationship constraints are met when calling methods and to extend assignment compatibility to use those type relationship constraints. This requires additions to `mono_method_verify` to check constraints on `call`, `calli` and `callvirt` instructions, this can probably be done in `do_invoke_method`. It also requires additions to `verify_stack_type_compatibility_full` to check the new assignment compatibility rules.

3.3.2 corlib

The core library needs to be changed to support type equality. Specifically the reflection library defined in mcs/class/corlib/System.Reflection

.Emit

. This API is predominately implement via internal calls that are then defined in mono/metadata/icall-def.h.

3.3.3 Cecil and IKVM

Mono.Cecil and IKVM are well known libraries for working with CIL code. They are generally considered better than the core library System.Reflection and System.Reflection.Emit. Any extensions of the system would be well served by extending these libraries to understand the extension as well.

3.4 Testing

As part of any implementation a number of test cases should be developed to show that the system works. These tests need to show that there is no CLI code that does not run in the new system (backwards compatibility) and that the new system is correct.

Normally these tests would be written in CIL assembly, however writing tests in CIL assembly would be time consuming and error prone. Instead test writing should be assisted by an assembly rewriter. The rewriter will look for calls to special methods and attributes in a CLI assembly and rewrite the bytecode and metadata to use features in the extended system. Assembly rewriting will be done using the CLI metadata reflection, writing and rewriting via `System.Reflection` or another metadata editor such as `Mono.Cecil`. Any implementation will extend these objects to support the new metadata constructs. With this we can mark up standard C# or F# code with special methods and attributes and rewrite the resulting assemblies to include the new metadata.

This system of testing has a number of advantages. Firstly it allows us to write our tests in a high level language such as C# instead of low level CIL. The main benefit of writing in a high level language should be clear, it's much easier. However we also get another less obvious benefit, that we are able to test both imperative (C#) and functional (F#) code. This should mean coverage over most aspects of the CLI.

Secondly we can run the tests on the standard CLI, this gives us something to test against. Running in the standard CLI is not equivalent to running the rewritten program in the new CLI (which we will call CLI+), but does allow some deductions to be made.

For type equality there are two methods we need to define. `U Cast<T, U>(T obj)` that checks `T` and `U` are equal at runtime and casts `obj` to type `U` if they are; and `void EqualTypes<T, U>()` that will check that T and U are equal types at runtime. Both these methods will throw an exception `TypeEqualityException` if `T` and `U` are not equal.

The rewriter will search an assembly for uses of `Cast<T, U>` and `EqualTypes<T,U>`. Any use of `EqualTypes<T,U>` will be removed and the method metadata rewritten to include the new type equality tags. Any use of `Cast<T, U>` will also rewrite the metadata to include the new type equality tags and will also remove the call to `Cast`.

Listing 3.3: Cast

```
public static class TypeEquality
{
    public static U Cast<T, U>(T obj)
    {
        if(typeof(T) == typeof(U))
        {
            return (U)(Object)obj;
        }
        else
        {
            throw new TypeEqualityException();
        }
    }
    public static void EqualTypes<T, U>()
    {
        if(typeof(T) != typeof(U))
        {
            throw new TypeEqualityException();
        }
    }
}
```

A program P is the source code in a CLI language, such as C#. It may or may not make use of `TypeEquality.Cast` and `TypeEquality.EqualTypes`. It can be compiled by a compiler C to give a CIL assembly, this assembly can be run on a runtime to give a value. The two runtimes are CLI and $CLI+$, values are either some value v representing the overall act of computation done by the program or an exception thrown by the program, the only exceptions we care about are *TypeEqualityException*, or *VerificationException*. Finally a CIL assembly can be rewritten by the rewriter R to produce a CIL+ assembly.

If R and CLI+ are correct then the following statements should hold:

Lemma 1. $C(P) \underset{CLI}{\rightarrow} TypeEqualityException \implies R(C(P)) \underset{CLI+}{\rightarrow} VerificationException$

Lemma 2. $R(C(P)) \underset{CLI+}{\not\rightarrow} VerificationException \implies C(P) \underset{CLI}{\not\rightarrow} TypeEqualityException$

Lemma 3. $C(P) \underset{CLI}{\rightarrow} v \wedge R(C(P)) \underset{CLI+}{\not\rightarrow} VerificationException \implies R(C(P)) \underset{CLI+}{\rightarrow} v$

Lemma 4. $C(P) \underset{CLI}{\rightarrow} v \iff C(P) \underset{CLI+}{\rightarrow} v$

Lemma 1 states that if a program throws a `TypeEqualityException` in the standard CLI then it will throw a `VerificationException` in the new CLI. If CLI+ doesn't throw a `VerificationException` then we know something is wrong with either the rewriter or the new runtime. For the sake of these tests we will assume that the rewriter is correct. This property does not hold in reverse ($R(C(P)) \underset{CLI+}{\rightarrow} VerificationException \implies C(P) \underset{CLI}{\rightarrow} TypeEqualityException$) as the call to `Cast<T, U>` or `EqualTypes<T, U>` might not be hit be every control flow path.

Lemma 2 states that if the rewritten program runs in CLI+ and does not throw a `VerificationException` then running the code in the CLI will not throw a `TypeEqualityException`. If a `TypeEqualityException` is thrown then something is wrong with the new runtime.

Lemma 3 states that if the program computes a value v in the standard CLI and verifies correctly in the new CLI then the new CLI should compute the same value v.

Lemma 4 states that if the same CLI assembly (no rewriting) is run on both runtimes they should compute the same value. While this is similar to lemma 3 it's difference is the lack of any assembly rewriting.

All this together means that we can do some verification of our new system against the old system. If a program ran correctly in the old system it should also run correctly in the new system (lemma 4). If a program has correct equality type constraints then it should run in the old system (and by the lemma 4 also run in the new system) and when rewritten it should run in the new system (lemma 3). If a program throws a `TypeEqualityException` then it should throw a `VerificationException` when rewritten and run in the new system.

3.5 Testing without a full implementation

While the above test plan is appropriate for checking an implementation is correct we do not have a real implementation. However we still wish to show that the system could work and is statically checkable. To do this we will be using a similar system to above. We can continue to write tests as described above, however instead of using an assembly rewriter to transform them into an extended CIL format we simply perform the checks that the extended verifier would perform. Thus for every method call we check to see if it calls `EqualTypes` and if so check that we can satisfy those constraints, and for every call to `Cast` we check that the two types are indeed equal either directly or via equality constraints.

Chapter 4

Values for type parameters

4.0.1 Concept of equality

To be able to say if a type `T` that depends on a value of `U` (that is `T<U a>`) is equal to `T<U b>` requires us to say what it means for `a` and `b` to be equal. All CLI objects have a method `bool Equal(object obj)` which we could use, however this is clearly unsound as an implementation of `Equal` could return nonsense, or never return at all.

Instead we propose using structural equality. That is two values of any reference type are equal if and only if they are the same reference, and two values of a value type are equal if and only if all their fields are equal in this manner as well.

In practice this amounts to checking that the values have the same bytes in memory. At runtime this is an easy check to make and dynamic type checks and reflection would have no issues with it. Doing this at compile time requires that each instantiation of a value parameter is constant, for value types this is simple to achieve while still being easily usable. Two separate instantiations of a type `T<int i>` with the same `int` value will both end up with the same byte values for the `int` value (although how and where this value is stored still needs to be discussed). For custom value types some extra work is required to make sure they are immutable but instantiation remains simple.

However given some reference type `U` with a constructor `U(int i)` two separate instantiations of a type `T<U u>` done via two `new` expressions will result in two distinct references. Given that all these values must be constant at compile time it's not clear how this could work. We can't reference the value from another type specification as that itself requires the reference.

Listing 4.1: An issue with reference values

```
var myT = new T<new U(1)>();
var myOtherT = new T<T<new U(1)>.u>();
}
```

We could extend the definition of `literal`(`const` in C#) to allow constants of custom types this would allow us to initialize and store the reference once and then use it in type parameters.

Listing 4.2: Literal references

```
const U U1 = new U(1);

public T<U1> SomeMethod(T<U1> t)
{
    return t;
}
```

While this seems ideal due to the way `literal` works this would cause significant difficulty for reference types, but it would work for value types and would solve the storage issue mentioned earlier, so still a wanted extension. The reason for the difficulty is that uses of `literal` fields replace the use site with the value at compile time. The value for a reference is a managed reference pointer, deciding where that pointer should point is impossible at compile time

4.0.2 Immutability

Type preservation means that an expression's type should not change under evaluation, therefore value type parameters should be immutable. As shown in subsection 2.1.4 the CLI does not have strong support for immutability. As such our initial work will concentrate on using the primitive types as their immutability can be easily guaranteed.

As noted above the values used as parameters should be immutable, allowing these values to change at runtime would be unsound as the system has already made judgements based on the static value. For primitive types immutability is easily achieved by making it impossible to write to or take the address of the parameter. This also covers us for custom value types, as although instance methods on value types take a managed pointer to the value (which could then be used to write to the fields) there is no way to get this reference in the first place. The value must be copied to a local variable and then instance methods can be called on that.

For references the value we care about is the reference itself, the object pointed to does not need to be immutable.

4.0.3 Operations

Once we have the ability to mark up types with values, we will want to use the value for operations. Either for changing the value before passing it on as another type parameter or for using at runtime. Support for using the value in normal methods at runtime seems trivial, just expose it similar to a static readonly field. However supporting the ability to do operations on value parameters before passing them to another type constructor is more challenging. Firstly it will require some effort to fit into the CIL bytecode, currently opcodes are only allowed in method bodies, we would have to either point to a method to calculate the operations on value parameters or find some other way to fit opcodes at the declaration level. Secondly we have the issue of soundness as user defined operations can do anything.

Chapter 5

Evaluation

Chapter 6

Conclusion

We've presented a specification for type relationship constraints in the CLI as well as as description of how theses could start to be implemented in the Mono runtime. We have also discussed ideas on how the CLI could be extended to support values as type parameters and the issues and advantages these would bring.

6.1 Future work

There are a number of ways to further this work, most obviously

Bibliography

[1] Agda. `http://wiki.portal.chalmers.se/agda`.

[2] Boost. `http://www.boost.org`.

[3] Configurable math library. `http://www.cmldev.net`.

[4] Coq. `http://coq.inria.fr`.

[5] F#. `http://fsharp.org`.

[6] Idris. `http://idris-lang.org`.

[7] Java: Type erasure. `http://docs.oracle.com/javase/tutorial/java/generics/erasure.html`.

[8] Scala. `http://www.scala-lang.org`.

[9] What is meant by scala's path-dependent types? `http://stackoverflow.com/questions/2693067/what-is-meant-by-scalas-path-dependent-types`.

[10] Ecma-335 common language infrastructure (cli), 2012.

[11] Lennart Augustsson. Cayenne - a language with dependent types.

[12] Kim B Bruce, Martin Odersky, and Philip Wadler. A staticlly safe alternative to virtual types, 1998.

[13] Andrew Kennedy and Claudio V Russo. Generalized algebraic data types and object-oriented programming, 2005.

[14] Microsoft. Units of measure (f#). `http://msdn.microsoft.com/en-us/library/dd233243.aspx`.

[15] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[16] Tim Sheard, James Hook, and Nathan Linger. Gadts + extensible kinds = dependent programming, 2005.

[17] Don Syme, Nick Benton, Simon Peyton-Jones, and Cedric Fournet. Proposed extensions to com+ vos, 1999.

[18] Wikipedia. Mars climate orbiter. `http://en.wikipedia.org/wiki/Mars_Climate_Orbiter`.