Imperial College London

Interim Report

Value dependent types for the CLI

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

Supervisor:
Professor Sophia Drossopoulou

January 16, 2013

Contents

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 solutions to be succinctly expressed at the CLI level and exposed to these languages.

Chapter 1

Introduction

1.1 Value dependent types

Dependent types allow static typing of expression based on values rather than just other types. There are some functional languages such as Agda[1], Coq[4], Idris[6] and Cayanne[13] 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 currently, `Components` is a list of component indices 0, 1, 2 etc.

```
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[1]

The CLI 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. To investigate the use and benefits of value dependent typing.

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

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. The CLI runs Common Intermediate Language (CIL) byte code. CIL is a type rich, stack based assembly language.

```
.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.

While CIL is targeted by a variety of languages Visual Basic and C# match it's semantics most closely so 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.

2.1.2 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` would be a value type while `Foo*` would be a reference type, the semantics are similar.

2.1.3 Literal and initonly

Fields in the CLI can be marked as `initonly`, and if they are static fields, `literal`.

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# shows a readonly Pair field being mutated, this 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.

2.1.4 Properties

As pointed out above the CIL has instructions for field access but it also has first class support for properties. 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` at `L_000f` 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 as such.

```
.class private auto ansi beforefieldinit Square
extends [mscorlib]System.Object
{
    .method private hidebysig static void Main() cil managed
    {
        .entrypoint
        .maxstack 2
        .locals init ([0] class Square sq)

        newobj instance void Square::.ctor()
        stloc.0
        ldloc.0
        ldc.i4.4
        stfld int32 Square::Length
        ldloc.0
        callvirt instance int32 Square::get_Area()
        call void [mscorlib]System.Console::WriteLine(int32)
        ret
    }

    .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
    }
}
```

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.

```
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
}
```

2.1.5 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 [23] 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 [23], the paper does give an insight into what we need to be thinking about while designing value parametrics. 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 [23] 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>[1]. 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

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

`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
{ ... }
```

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[24].

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[20]

> "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.

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. [14] 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[13] 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 [13].

```
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 that reduce some constraints on data type constructors, in particular they allow pattern matching and recursion in a data constructor. The common example is a type for terms in a small language, without GADTs you cannot have the type checker check that the expression trees are correct.

Listing 2.1: ADT

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

It's clear that the first expression passed to `Cond` must evaluate to a boolean result (from `Equals`), but the type system cannot express that. If we add GADTs to our language we can rewrite `Exp` to the following.

Listing 2.2: 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 a boolean expression as the first argument to `Cond`. 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 [19] shows how GADT programs can be expressed in C# with some modifications to the language. The two modifications proposed by [19] 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 [19] 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.

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 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 [19]).

2.8 Conclusion

Having looked at all these type systems we can see a progression of expressivity, and in some cases equality.

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 was **value dependence**, this 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 [21] 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. We've explored how GADTs can be expressed in C# with some modifications. Taking these ideas and applying them to the CLI is easily possible. Generic constraints are already stored in the metadata, 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.

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

	Turing complete	Type safe printf	Type sized lists	Decidable	Units of measure	Path dependence
Templates	Yes	Yes	Yes	No	Yes	?[2]
Value dependence	No	Yes	Yes	No	Yes	Yes
GADT	No	Yes	Yes	Yes	No[3]	No
Generics	No	No	No	Yes	No	No

2.8.1 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 and reference 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.

2.8.2 Immutability

Type preservation means that an expressions type should not change under evaluation, so value type parameter should be immutable. As shown in subsection 2.1.3 the CLI does not have strong support for immutability. As such our initial work will concentrate on using the primitive types as there immutability can be easily guaranteed.

2.8.3 Operations

Once we have the ability to mark up types with values we will want to use the value for operations. Support for using the value in normal methods 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.

Chapter 3

Project plan

As already briefly mentioned in the introduction this project can be split into three major parts, investigation, design and implementation.

3.1 Investigation

The first part of the project is an investigation into value dependent types and similar systems. This has already been covered in our background research. From this we have an understanding of how these systems are useful and how they can be designed and implemented and this will guide us on the design of the CLI extension. This investigation forms the background research part of the final report and has mostly been done as part of this interim report.

3.2 Design and implementation

The second part of the project is to design and implement extensions to the CLI. For each extension we will show what changes need to be made to ECMA-335 to support the extension, and implement the extension in the open source CLI Mono. We list each of these and propose some detail of it's implementation, all syntax is highly subject to change however.

3.2.1 GADTs

The first extension will be to add type equality constraints and a match and bind instruction to the CLI. To do this we will take the ideas from [19] and translate them to apply to the CLI.

Listing 3.1: Type equality constraints in extended C#

```
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>()
    {
        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>()
    {
        return Head.Append(Tail.Flatten<U>());
    }
}
```

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>()
        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
}
```

match		
Format	Assembly Format	Description
0xFE 0x19	match typetoken	matches and object against an open or close generic type and returns the object cast to that type and the RuntimeHandles for the types required for closure if typetoken is an open generic type.

Stack Transition: . . . , obj → . . . , obj, RuntimeHandles

3.2.2 Value parameters

The second extension is to add primitive value parameters to types and methods. This extension will only allow primitive types as parameters and has no support for constraints or operations on parameters at compile time.

Listing 3.3: Value parameters in extended C#

```
public class Value<int value>
{
    public static int Dup<int v>(Value<v> a, Value<v> b)
    {
        return Value<v>.value * 2;
```

```
    }

    public static void Print()
    {
        System.Console.WriteLine(value);
    }
}
```

Listing 3.4: Corresponding CIL

```
.class public auto ansi beforefieldinit Value``1<int value>
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 static int32
        Dup``1<int v>(class Value``1<$$0> a, class Value``1<$$0> b) cil managed
    {
        .maxstack 2
        .locals init (int32 temp)
        ldsfld int32 Value``1<$$0>::value
        ldc.i4.2
        mul
        ret
    }

    .method public hidebysig static void Print() cil managed
    {
        ldsfld int32 Value``1<$0>::value
        call void [mscorlib]System.Console::WriteLine(int32)
        ret
    }
}
```

3.2.3 Value constraints and operations

The third extension is to add constraints and operations to value parameters. It's unclear at this stage quite how this could work in CIL and so we don't provide an example of it.

3.2.4 User defined types as value parameters

The final extension is to allow value parameters to be values of user defined types, not just primitive types.

Listing 3.5: User defined types as value parameters in extended C#

```
public class Value<MyClass c>
{
    public static int Access<MyStruct s>()
    {
        return s.field;
    }

    public static void Print()
    {
        System.Console.WriteLine(c.property);
    }
}
```

Listing 3.6: Corresponding CIL

```
.class public auto ansi beforefieldinit Value''1<class MyClass c>
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 static int32
        Access''1<MyStruct s>() cil managed
    {
        .maxstack 2
        ldvalue $$0
        ldfld int32 MyStruct::field
        ret
    }

    .method public hidebysig static void Print() cil managed
    {
        ldvalue $0
        callvirt instance int32 MyClass::get_property()
        call void [mscorlib]System.Console::WriteLine(int32)
        ret
    }
}
```

3.3 Timetable

1. GADT extensions - 25^{th} February

2. Primative value parameters - 25th March
3. Value parameter constraints and operations - 22nd April
4. User defined types as value parameters - 20th May
5. Further improvements and formalization - 17th June

Chapter 4

Evaluation plan

4.1 Semantics

Defining semantics for what value dependence means can be done in two ways. Firstly we could extend the ECMA specification ([10]). Secondly we could take a formal specification of the CLI and extend that. While work has been done on formalization of CLI languages such as C#, work on formalizing the CLI does not seem to have been done. If we can work out how value dependence should work in the CLI then extending the ECMA specification is a required aim, as it is the basis for compiler writers targeting the CLI. Extending a formal specification would be a stretch goal to complete once other goals are achieved, both because it may require translating our extension to C# to use a lightweight C# formalization based on featherweight GJ and secondly as a formalization is not required for the implementation work.

4.2 Implementation

Given an extension to the CLI specification we want to show that the extension can be implemented. To do this we will extend the open source Mono run time to support value dependent types. The most important aspect is correctness but performance should be kept in mind. As pointed out in section 1.2.1 we want certain performance characteristics out of the system.

Chapter 5

References

[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] Philippe Altherr and Vincent Cremet. Inner classes and virtual types, 2005.

[12] Nada Amin. Dependent object types.

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

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

[15] Dave Clark, Sophia Drossopoulu, James Noble, and Tobias Wrigstad. Tribe: More types for virtual classes.

[16] Collin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism, 2012.

[17] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord.

[18] Andrew Kennedy. Types for units-of-measure: Theory and practice, 2009.

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

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

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

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

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

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