Value dependent types for the CLI

Fraser Waters*

11/05/12

Contents

*fraser.waters08@imperial.ac.uk

Part I
Introduction

1 Motivation

One of the main motivations for looking into value dependence is for numeric types such as Vector3, Vector4, Matrix3x4. These are used for graphics and physics applications where most vectors and matrices in the problem domain are small (3 or 4 elements). 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. The following shows the code used to generate all the required dot product functions currently, and how I imagine it might look with value dependence.

```
WriteLine("/// <summary>");
WriteLine("/// Calculates the dot product (inner product) of two vectors.");
WriteLine("/// </summary>");
WriteLine("/// <param name=\"left\">First source vector.</param>");
WriteLine("/// <param name=\"right\">Second source vector.</param>");
WriteLine("/// <returns>The dot product of the two vectors.</returns>");
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}]",
WriteLine("return {0};", dotproduct);
```

```
Dedent();
WriteLine("}");

public static float Dot<int n>(Vector<n> a, Vector<n> b)
{
        float dot = 0;
        for(int i=0; i<n; ++i)
        {
                dot += a[i]*b[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.

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

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

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);
```

2 Performance

Currently we can we can define an interface `Vector` that defines the indexing operator and length property, and then write functions useing this interface. This saves us the effort of creating and maintaining the generator but at the cost of runtime performance.

```
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 (and this give some suggestions as to how we would want to implement dependent types) is that it adds an extra instance variable to each vector and means 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. With dependent types we could write code in a similar way but with better performance characteristics.

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.

3 Value dependent types

Dependent types allow static typing of expression based on values rather than just other types. While some functional languages such as Agda, Coq and Idris support dependent types dependent types are not so common in object oriented languages. While fully general value dependent types are rare some weaker versions, including path dependent types and virtual types, are used in some mainstream languages. Notably Scala supports both path dependence and virtual types, F# supports units of measure allowing numbers to be typed based on a unit value, and C++ has templates that can be parametrized on values.

4 The CLI[1]

The CLI is a specification for a virtual execution environment, that is implemented by Microsoft's CLR (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 Java Virtual Machine (which for example has no concept of generic types despite Java supporting them). *Why target the CLI? Finding it hard to phrase this. But two reasons, firstly because I prefer the CLI tech to Java the only other big VM, and because extensions to the CLI can then be used by languages targeting the CLI opening up easy extension opportunities to many languages rather that say adding dependent types to just C#.*

The ability to retain high level type information allows easy interoperability between separate CLI modules, even with modules compiled using different languages. This feature starts to fall apart when languages add typing extensions that aren't supported by the CLI. Units of measure in F#, for example,

[1] Common Language Infrastructure

are erased at compile time meaning that other modules which consume an F# module where units of measure were used cannot see 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 can be written in terms of dependent types then we can *fix* them, else at least our extension does not suffer the same problem of interoperability. Of course 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 bytecode and the speed to process it and the speed and size of the JITed code.

5 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.
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.

Part II
Background

6 The CLI

6.1 Common intermediate language

The CLI runs Common Intermediate Lanugage (CIL) bytecode. 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 the 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 CIL instructions.

6.2 Generics

The CLI supports parametric polymorphic types via generics types that are parametrized on other types (value dependence would allow types to also be parametrized on values). The MSR White paper [6] 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 differs slightly from this paper it gives an insight into what we need to be thinking about while designing value parametric. It's worth taking some time to look at how generics ended up being specified in [4] and implemented in Mono (due to copyright reasons we can't look at Microsofts open source CLR code).

Generics are defined in section II.9 of [4]. 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 [6] 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 $List\langle T\rangle$ while retaining run time information such that the run time type of $List\langle object\rangle$ is different to $List\langle int\rangle$. $List\langle int\rangle$ is also special in that int is a value type and yet the run time can use a $List\langle int\rangle$without causing excessive boxing of values.

If we look at the definition of$List\langle T\rangle$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
2 extends 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...List*`1 declares a new class type with one generic parameter, which has no constraints. The implements clause lists a number of interfaces which *List* implements, the first three of these interfaces are themselves generic. On line 3 the *System.Collections.Generic.IList*`1 syntax indi-

cates that we mean the generic *IList* with one parameter '1, $\langle !0 \rangle$ refers to the first generic class paramter, and passes that as the type argument to *IList*.

7 C++

Uses of value templates in C++ looking at the standard library, Boost and CML (the Configurable Math Library). The standard C++ library uses value templates in a few places including $std :: ratio$, the random number generation library and *type_traits*. The open source Boost[1] 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 $Array \langle T, N \rangle$, but also scattered throughout the other libraries. For example in $Spirit :: Qi$, a parser combinator library, the type *uint_parser* is templated on the typename 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, vector is templated on a typename *ElementT* and *StorageT*. Moreover two of the built in storage types (fixed and external) are templated on the value of how many elements they store. When using these statically sized storage types you get extra static type safety that you're not mixing vector sizes in operations.

8 F#

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 because of a mismatch between Imperial and Metric units in force calculation, a very expensive mistake as the craft cost $125 million. Units of measure is done at compile time and all units information is erased from the run time, this means that values cast to *Object* cannot be recast to a measured type safely at run time, but also that these units cannot be exposed as part of an public interface to be consumed by other CLI languages such as C# or VB.

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

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

Units of measure 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 $float \langle meter \rangle$.

```
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[7]

> "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.

9 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[8].

```
1  case class Board(length: Int, height: Int)
2  {
3    case class Coordinate(x: Int, y: Int)
4    {
5      require(0 <= x && x < length && 0 <= y && y < height)
6    }
7    val occupied = scala.collection.mutable.Set[Coordinate]()
8  }
9
10 val b1 = Board(20, 20)
11 val b2 = Board(30, 30)
12 var b3 = b1
13 val c1 = b1.Coordinate(15, 15)
14 val c2 = b2.Coordinate(25, 25)
15 b1.occupied += c1
16 b2.occupied += c2
17 b3.occupied += c1
18 // Next line doesn't compile
19 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.

```
1  class Board
2  {
3    int length, height;
4
5    public Board(int length, int height)
6    {
7      this.length = length;
8      this.height = height;
9    }
10
11   class Coordianate<Board b>
12   {
13     public Coordinate(int x, int y)
14     {
15       require(0 <= x && x < b.length && 0 <= y && y < b.height)
16     }
17   }
18
19   Set<Coordianate<this>> occupied = new Set<Coordinate<this>>;
20 }
```

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

10 Virtual types

Virtual types are also found in Scala, it allows 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.

```
1  class A
2  {
3    type T
4    abstract T foo();
5  }
```

```
6
7  class B
8  {
9    override type T = String
10   override T foo() { return "string"; }
11 }
```

While virtual types can be useful everything they accomplish can also be done with generics, albeit with sometime much more syntax.

11 First class types

Cayenne[2] is a language with support for dependent types and first class types (i.e. types can be be used like values). Cayenne is a functional language inspired by Haskell, it's unlikely we can straight lift ideas 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 [2].

```
1  printfType :: String -> #
2  PrintfType "" = String
3  PrintfType ('%':'d':cs) = Int    -> PrintfType cs
4  PrintfType ('%':'s':cs) = Stirng -> PrintfType cs
5  PrintfType ('%':_:cs)   =           PrintfType cs
6  PrintfType (_:cs)       =           PrintfType cs
7
8  printf :: (fmt::String) -> PrintfType fmt
9  printf fmt = pr fmt ""
10
11 pr :: (fmt::String) -> String -> PrintfType fmt
12 pr "" res = res
13 pr ('%':'d':cs) res = \(i::Int) -> pr cs (res ++ show i)
14 pr ('%':'s':cs) res = \(s::String) -> pr cs (res ++ s)
15 pr ('%':c:cs) res   = pr cs (res ++ [c])
16 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 $*$).

12 Generalized algebraic data types

Generalized algebraic data types (GADTs) are predominatly a feature of functional languages. The use of GADTs in object orientated langauges is less common but [5] shows how all GADT programs can be expressed in C# and that with some small modifcations to the langauge be easily supported.

Part III
Project plan

As already breifly mentioned in the introduction this project can be split into three major parts.

13 Investigation

The first part of the project is an investigation into value dependent types and similar systems. This has already been covered somewhat in our background research. An understanding of how these systems are usefull and how they can be designed and implemeneted will be used to guide us on the design of the CLI extension.

14 Design

The second part of the project is to design an extension to the CLI that supports dependent types. Exactly what would be supported in this new system will depend on where are investigation has taken us. We will show what changes need to be made to ECMA-335 to support the extension.

15 Implementation

The last part of the project is to implement the extension. We will be using the open source Mono project for this. This will require us to change both the runtime and the assembler, to support the new syntax.

Part IV
Evaluation plan

16 Semantics

Defining semantics for what value dependence means can be done in two ways. Firstly we could extend the ECMA specification, which is written in formal specification speak. Secondly we could take a formal specification of the CLI and extend that. If we can work out how value dependence should work in the CLI then extending the ECMA specification is a required aim. Extending a formal specification would be a stretch goal to complete once other goals are achieved.

17 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 2 we want certain performance characteristics out of the system.

[1] Boost.

[2] Cayenne - a language with dependent types.

[3] Configurable math library.

[4] Ecma-335 common language infrastructure (cli).

[5] Generalized algebraic data types and object-oriented programming.

[6] Proposed extensions to com+ vos.

[7] Units of measure (f#).

[8] What is meant by scala's path-dependent types?