Value dependent types in the CLI

Fraser Waters

Value dependence

- Type parameterized by a value
- Similar to how parametric polymorphism
- Used in Agda, Coq and some other functional languages
- Not very common in mainstream languages
- Except C++ templates

The CLI

- The Common Language Infrastructure
- A specification for a virtual execution environment (VES)
- Implemented by Microsoft’s Common Language Runtime (CLR) and the open source Mono project
- Target machine for VB, C#, F#, IronPython and more

Types and the CLI

- Strongly and statically typed
- Retains a high level of information
  - JVM has no concept of generics, Java does!
- Easy interoperation
  - Even different languages
- Not perfect!
  - Some languages features don’t map to CLI

Motivation

- Can catch more problems at compile time
  - If it compiles it works
- Increase coverage of advanced features
- Performance

Background research

- C++ templates

C++ templates

```
cml::vector<float, fixed<3>> a(1, 0, 0);
cml::vector<float, fixed<2>> b = a;

cml::matrix<float, fixed<2, 2>>
  i(1, 2, 3, 4);
cml::matrix<float, fixed<3, 3>> j = i;
```

Background research

- C++ templates
- F# units of measure

F# units of measure

```
let speed = 55.0<meter/second>
let time = 3.5<second>
let distance = speed *  time
let garbage = speed + time
speed : float<meter/second>
time : float<second>
distance : float<meter>
```

Background research

- C++ templates
- F# units of measure
- Path dependent types

Path dependent types

```
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
b1.occupied += c2
```

Background research

- C++ templates
- F# units of measure
- Path dependent types
- Virtual types

Virtual types

```
class A
{
  type T
  abstract T foo() ;
}
class B
{
  override type T = String
  override T foo() { return “string”; }
}
```

Background research

- C++ templates
- F# units of measure
- Path dependent types
- Virtual types
- First class types

First class types

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

Background research

- C++ templates
- F# units of measure
- Path dependent types
- Virtual types
- First class types
- Generalized Algebraic Data Types

Generalized Algebraic Data Types

```
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

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

GADTs via type equality

- GADTs are equivalent to generics and type equality constraints
- CLI already supports generics
- Add type equality constraints get GADTs as well

Type equality constraints

- Type equality constraints allow us to add constraints to methods of the form T=U
- T and U are any valid type reference
- Can have multiple constraints
- Constraints used to augment assignment compatibility

Type equality constraints

```
// Once a method has a constraint T=U and variables of type T
// can be treated as U and vice versa
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 head.Append(tail.Flatten());
   }
}
```

Specification

- Add where clause to methods
- Enhance assignment compatibility with type equality
- Check constraints before calling methods
- Add constraints to a new Metadata table

Type equality constraints

- Proof of concept demo
- Special methods to mark constraints
  - Standard practice (e.g. Code contracts)
- Methods perform dynamic check at runtime
- Constraint checker can check statically

```
public abstract class List<T> {
   public abstract List<T> Append(List<T> list) ;
   public virtual List<U> Flatten<U>( )
   {
       EqualTypes<T, List<U>>();
   }
}
public class Nil<T> : List <T> {
   public override List<U> Flatten<U>() {
       EqualTypes<T, List<U>>();
       return new Nil<U>;
   }
}
public class Cons<T> : List<T> {
   T head; List<T> tail;
   public override List<U> Flatten<U>() {
       EqualTypes<T, List<U>>();
       return Cast<T, List<U>>(head).Append(tail.Flatten());
   }
}
```

DEMO

Conclusion

- Demo shows that checking these constraints statically is possible
- Small addition to the CLI specification

Values as type parameters

- Types parameterized on values

```
Vector<3>
Matrix<4,4>
Json<“““{ “name”:”example” }”””>
Float<Meters>
Float<Meters / Second>
```

Motivation

- Physics and graphics work
- Often working with small fixed size vectors and matrices
- Want to be able to define the Vector type just once for any size
- Currently not possible in an efficient way
- Resorted to pragmatically generating 10s of different types

Current state

```
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);
Indent("{");
var dotproduct = string.Join(" + ", Components.Select(component =>
    string.Format("left.{0} * right.{0}", component)));
WriteLine("return {0};", dotproduct);
Dedent("} ");
```

What we want

```
/// <summary>
/// Calculates the dot product (inner product) of two vectors.
/// </summary>
/// <param name="left">First source vector.</param>
/// <param name="right">Second source vector.</param>
/// <returns>The dot product of the two vectors.</returns>
public static float Dot<int n>(Vector<n> left, Vector<n> right)
{
    var result = 0;
    for(int i=0; i<n; ++i)
    {
        result += left[i] * right[i];
    }
    return result;
}
```

Client code

- Either way client code looks similar

```
var a = new Vector3(0, 1, 2);
var b = new Vector3(3, 4, 5);
var dot = Vector.Dot(a, b);

var a = new Vector<3>(0, 1, 2);
var b = new Vector<3>(3, 4, 5);
var dot = Vector.Dot(a, b);
```

Issues

- Equality

Equality

- What does it mean for values to be equal?
- User defined "`Equals`" operator?
  - Unsound, have to run user code at compile time
- Byte equivalent?
  - Ok for value types, but what about references?

Equality

```
class U { ... }

var myT = new T<new U(1)>();

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

Equality

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

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

Issues

- Equality
- Immutability

Immutability

- CLI has weak rules for immutability
  - `readonly` and `literal`
- For value types simple
  - Disallow field writes
  - Disallow taking the address
- For references even simpler
  - Only care about the identity of the reference
  - Just disallow field writes

Issues

- Equality
- Immutability
- Operations

Operations

```
public class Array<T, int length> {
...
public
  Array<T, length+n> // return type
  Concatenate<int n> // method name
  (Array<T, n> other ) // parameters
  { ... } // body
}
```

Issues

- Equality
- Immutability
- Operations
- Variable runtime size

Variable runtime size

```
public struct Vector
{
  public readonly float[] values;

  public Vector (int n) {
    values = new float[n];
  }
  ...
}
```

Variable runtime size

```
.class sequential ansi sealed nested public beforefieldinit Vector3
    extends [mscorlib]ValueType
{
    .field public valuetype Vector3/<Values>e__FixedBuffer0 Values
    {
          .custom instance void
                    [mscorlib]FixedBufferAttribute::.ctor(class [mscorlib] Type, int32) =
                              { type(float32) int32(3) }
    }

    .class sequential ansi sealed nested public beforefieldinit  <Values>e__FixedBuffer0
          extends [mscorlib]ValueType
    {
          .custom instance void [mscorlib]UnsafeValueTypeAttribute::.ctor()

          .custom instance void [mscorlib]CompilerGeneratedAttribute::.ctor()

          .field public float32 FixedElementField
    }
}
```

Conclusion

- Dependent typing brings a lot of problems
- Trade off between performance and elegance

Evaluation

- Type equality constraints
  - Small addition with large reach
  - Mono is complex
- Value dependent types
  - Understand why they're not used
  - But would solve a lot of problems
- Overall
  - Time management should of been better

Thank you

Questions?