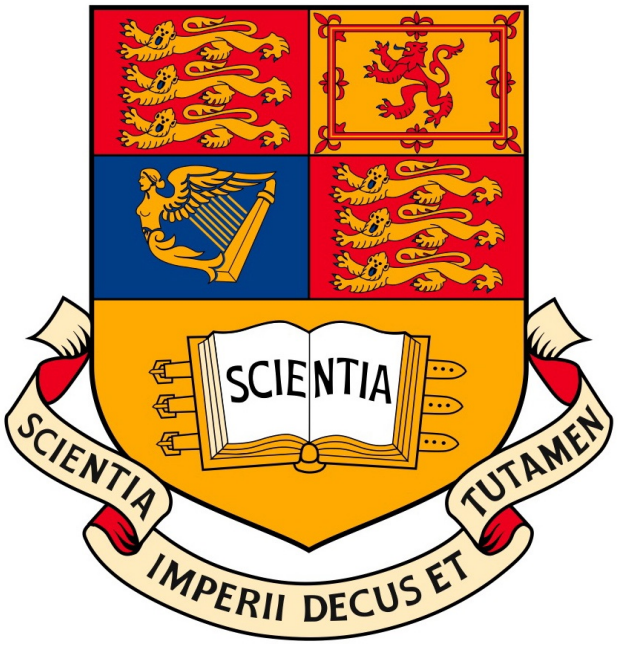

Imperial College London

Value dependent types for the CLI

Type equality constraints

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

Supervisor:
Professor Sophia Drossopoulou

June 12, 2013

Contents

Abstract

Type equality constraints would allow the CLI to better express generalized algebraic data types.

Chapter 1

Introduction

1.1 Generalized algebraic data types

This report looks into how to add type equality constraints to the CLI. Type equality constraints for C# are described in [2], we'll 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 [2] 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.

1.2 Example

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

Listing 1.1: 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 1.2: 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
}
```

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.

Chapter 2

Specification changes

2.1 Generic constraints

Section II.9.11 (Constraints on generic parameters)[1] 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 seperate 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 neccessry 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.

2.2 Assignment compatability

Assigment compatability is defined in section I.8.7 of [1], further to this verification assigment compatability is defined in III.1.8.1.2.3. Verification assgiment compataiblity is mostly defined in terms of general assigment compatability from I.8.7.3.

Verification assigment compatability is used be the verifier for determining if method calls, field referances and loads and stores are valid for a given type and signature. If verification assingment compatability is extended to understand type eqaulity 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.

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 enviroment Γ and the equality constraints on the current method ϵ which defines equal types.

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

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

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

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

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

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

2.3 Method calls

Any method calls must be checked that any equality constraints on the method being called can be satisifed 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 verifiabilty rules for the call, calli and callvirt instructions.

2.4 Metadata tables

Metadata tables are specified in section 22 (Metadata logical format: tables). We're mostly intreasted 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)

Chapter 3

Implementation changes

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 TypeRelationshipConstraint table is very similar to the GenericParamConstraint table, the current code for GenericParamConstraint can guide the new code for TypeRelationshipConstraint.

Verification is done in verify.c. Additions need to be made to verifiy that type relationship constraints are met when calling methods and to extend assingment compatability to use those type relationship constraints.

3.2 corlib

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

.Emit

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

3.3 Cecil and IKVM

Mono.Cecil and IKVM are well known libaries 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 libaries to understand the extension as well.

Chapter 4

References

Bibliography

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

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