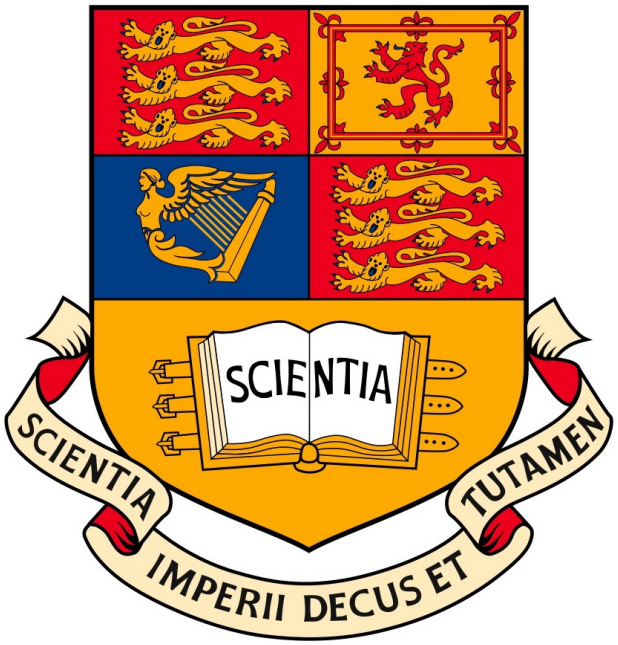

Imperial College London

Value dependent types for the CLI

Type equality constraints

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

Supervisor:
Professor Sophia Drossopoulou

January 21, 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

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 1.1: 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 1.2: 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]).

1.2 Implementation

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 1.3: 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 1.4: 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
}
```

Chapter 2

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`.