CS611 Lecture Leture 34 **Recursive Types** 21 November 2003

Scribe: Dan Williams, Revelino Guron Lecturer: Radu Rugina

We've been talking about types so we next consider type equivalence.

1 Type Equivalence

We want the equivalence relationship to have the properties of reflexivity, symmetry, transitivity. We then take the compatible closure of this relation, which propagates the equivalence of types down to components of more complex types. For instance, if $\tau_0 \equiv \tau_2$ and $\tau_1 \equiv \tau_3$, then $\tau_0 \rightarrow \tau_1 \equiv \tau_2 \rightarrow \tau_3$. For the typed lambda calculus with sums and product types, the equivalence relation is type equality.

One complication may arise in real languages, where the programmer can give names to types. For instance, in C, a declaration `struct foo {...}` assigns name `foo` to the type that this structure represents. Assigning names to types affects the equivalence relation. There are two alternatives, usually referred to as name equivalence and structural equivalence.

1. *Name equivalence:* With name equivalence, two types are equivalent if they have the same name, regardless of their structure. Consider the following example in C:

   ```
   struct { int x, y; } a, b;
   struct { int x, y; } c;
   a=b;   /* ok */
   c=a;   /* error */
   ```

   We see that even though the structures of the types are the same, they are not considered equivalent types and thus cause a type error. Here, the compiler automatically gives different names to the two structures.

2. *Structural equivalence:* With structural equivalence, two types are equivalent if they have the same structure. Consider the following example in ML:

   ```
   val a = {x=0, y=1};
   val c = {x=1, y=0};
   c=a;   /* ok */
   ```

   Here the types of `c` and `a` are equivalent, because they have the same structure, consisting of a pair of integers.

2 Recursive Types

Let us try to build a type for a recursive data structure such as a list. Such a type should look like this:

$$\tau = \mathsf{unit} + \mathsf{int} \times \mathsf{unit} + \mathsf{int} \times (\mathsf{int} \times \mathsf{unit}) + \ldots$$

Here unit is the type that describes an empty list. The above type for lists has an infinite structure, but has a regular pattern. To understand this pattern, let us see how one defines such a data structure in a real language. Consider a datatype definition for lists in ML:

```
datatype List = Empty | Cell of int * List
```

We can see that this is a recursive definition for `List`. Indeed, we can re-write our type for lists as:

$$\tau = \mathsf{unit} + \mathsf{int} \times (\mathsf{unit} + (\mathsf{int} \times \mathsf{unit}) + \ldots)$$

and see that this definition corresponds to the recursive equation:

$$\tau_{list} = \mathsf{unit} + \mathsf{int} \times \tau_{list}$$

2.1 The Recursive Type Operator

We formally describe recursive types using a recursive type operator μ, defined such that $\mu T.\tau$ is the fixed point solution of the recursive type equation $T = \tau$. Here, T is a type variable and τ is a type expression which contains free occurrences of T.

Using this operator, we can write our list type as

$$\tau_{list} = \mu T.\mathsf{unit} + \mathsf{int} \times T$$

If $\tau_0 = \mu T.\tau$ is a recursive type, then τ_0 models the solution of the equation $T = \tau(T)$. Hence, we should have:

$$\tau_0 \sim \tau(\tau_0) \ \sim \ \tau(\tau(\tau_0)) \ \sim \ \ldots$$

where the relation $\sim$ indicates indicates that two recursive types have the same (infinite) structure. In other words, we have the following types with the same structure:

$$\mu T.\tau \ \sim \ \tau[T \mapsto \mu T.\tau] \ \sim \ (\tau[T \mapsto \mu T.\tau])\ [T \mapsto \mu T.\tau] \sim \ldots$$

Essentially, this sequence "unrolls" the recursive type.

In our list example, the following two types have the same structure:

$$\mu T.\mathsf{unit} + \mathsf{int} \times T \ \sim \ \mathsf{unit} + \mathsf{int} \times (\mu T.\mathsf{unit} + \mathsf{int} \times T)$$

The question is whether we should regard two recursive types with the same structure $\mu T.\tau$ and $\tau[T \rightarrow \mu T.\tau]$ as equivalent (with respect to $\equiv$) or not? The two possible answers to this question correspond to equi-recursive and iso-recursive types.

2.2 Equi-recursive types

With equi-recursive types, a recursive type and its unrolled version are equivalent:

$$\mu T.\tau \ \equiv \ \tau[T \rightarrow \mu T.\tau]$$

This puts responsibility on the compiler to figure out if two types have an isomorphic structure. If we represent these recursive types as graphs, then this problem reduces to figuring out if the corresponding graphs are isomorphic. Consider the following two recursive types:

$$\tau_1 = \mu T.T \rightarrow \mathsf{int} \qquad \tau_2 = (\mu S.S \rightarrow \mathsf{int}) \rightarrow \mathsf{int}$$

To determine that these recursive types are equivalent, the compiler must figure out that the following graphs are isomorphic.

2.3 Iso-recursive types

Using iso-recursive types, the compiler treats a recursive type and its unrolled version as not being equivalent:

$$\mu T.\tau \not\equiv \tau[T \rightarrow \mu T.\tau]$$

Instead, the compiler expects that the programmers will explicitly write type annotations to move between these two types, using fold and unfold operations:

$$(\mu T.\tau) \quad \stackrel{\mathsf{unfold}}{\longrightarrow} \quad \tau[T \rightarrow \mu T.\tau]$$

$$\mu T.\tau \quad \stackrel{\mathsf{fold}[\mu T.\tau]}{\longleftarrow} \quad (\tau[T \rightarrow \mu T.\tau])$$

To model these operations, we extend our calculus with fold and unfold expressions, and with recursive types:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{fold}\ [\mu T.\tau]\ e \mid \mathsf{unfold}\ e \\ \tau & ::= & \ldots \mid \mu T.\tau \\ v & ::= & \ldots \mid \mathsf{fold}\ [\mu T.\tau]\ v \end{array}$$

The operational semantics for the new constructs are:

$$\frac{e \to e'}{\mathsf{unfold}\ e \to \mathsf{unfold}\ e'} \qquad \frac{e \to e'}{\mathsf{fold}[\mu T.\tau]\ e \to \mathsf{fold}[\mu T.\tau]\ e'} \qquad \frac{}{\mathsf{unfold}\ (\mathsf{fold}\ [\mu T.\tau]\ v) \to v}$$

The typing rules for the new constructs are:

$$\frac{\Gamma \vdash e : \tau[T \to \mu T.\tau]}{\Gamma \vdash \mathsf{fold}\ [\mu T.\tau]\ e : \mu T.\tau} \qquad \frac{\Gamma \vdash e : \mu T.\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau\ [T \to \mu T.\tau]}$$

In real languages, such as ML and C, fold and unfold are implicitly associated with other language constructs. In ML, unfold is represented as pattern matching, and folding is represented by applying a datatype construction. In C, unfolding is represented by dereferencing a pointer ($*x$), whereas folding is represented by taking the address of a variable ($\&x$).

2.3.1 Typing $\lambda x.\, xx$

Recursive types allow us to type the combinator *self* $= \lambda x.\, xx$. Recall that in the simply typed lambda calculus we were not able to type this expression because we couldn't find two types *tau* and τ' such that $\tau = \tau \to \tau'$. This is a recursive type equation, so now we can build a recursive type which satisfies this equation. Choose $\tau' = \mathsf{int}$. Then:

$$\tau = \mu T.T \to \mathsf{int}$$

Let's now give type τ to the variable x in the abstraction. When we try to type the application xx, the first x must be an abstraction. To change the type of x, which is currently τ, into an abstraction, we must apply the unfold operator:

$$(\lambda x : \mu T.T \to \mathsf{int}.\ .\ (\mathsf{unfold}\ x)\ x)$$

The type of the whole expression is now $\tau \to \mathsf{int}$. To turn this type back to τ we must apply the fold operator:

$$\mathsf{fold}[\mu T.T \to \mathsf{int}]\ (\lambda x : \mu T.T \to \mathsf{int}.\ (\mathsf{unfold}\ x)\ x)$$

2.3.2 ML Example

Next we will go through an example which shows where folding and unfolding occur implicitly in ML. Recall our definitions of `true, false, and` in lambda calculus.

$$\begin{array}{rcl} \mathit{true} & = & \lambda x.\,.\lambda y.\,x \\ \mathit{false} & = & \lambda x.\,.\lambda y.\,y \\ \mathit{or} & = & \lambda b.\,\lambda c.\,b\ b\ c \end{array}$$

One can easily write these in ML, by writing an abstraction $\lambda x.\, e$ as "`fn x => e`". So we can write these expression in ML as:

```
val chtrue  = fn x => fn y => x;
val chfalse = fn x => fn y => y;
val chor    = fn b => fn c => b b c;
```

However, compiling this program will fail because the compiler will not be able to infer types for the last expression, since `b` is applied to itself. The solution for this is similar to what we've seen in the previous example: build a recursive type for `b` and unfold `b` before applying to itself. We can build such a recursive type using a recursive datatype declaration:

```
datatype chbool = CH of chbool ->  chbool ->  chbool
```

Now we can rewrite `chor` as:

```
val chor = fn (CH b) => fn c => b (CH b) c;
```

This expression will successfully typecheck now. The pattern matching "`fn (CH b)`" in the argument of the abstraction performs the unfolding. The application of the type constructor in the function body "`b CH(b) c`" performs the folding operation.

Exercise. With the current definitions of `chtrue`, `chfalse`, and `chor`, the following expression will not typecheck in ML:

```
chor chtrue chfalse;
```

What changes are required so that ML will successfully typecheck this expression?