CS611 Lecture 30 | Recursive Types | November 6, 2000

Scribe: Cristina Patron and Suzanne Shontz | Lecturer: Andrew Myers

1 tF Example

In the last lecture, we finally arrived at a type-safe universal language named tF. Here are the formation rules for tF:

$$\begin{aligned} e &::= x \mid b \mid \mathsf{fn}\ x\!:\!\tau.e \mid e_1\ e_2 \mid e_1 + e_2 \mid \langle e_1, e_2\rangle \mid \mathsf{first}\ e \mid \mathsf{rest}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e_0\ e_1\ e_2 \\ &\quad \mid \mathsf{rec}\ y\!:\!\tau.\mathsf{fn}\ x.e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ \tau &::= B \mid \tau \to \tau_2 \mid \tau_1 * \tau_2 \mid \tau_1 + \tau_2 \end{aligned}$$

where e is an expression, and τ are the typed expressions.

The following is an example of a tF program:

$\mathsf{let\ factorial} = (\mathsf{rec\ fact}\!:\!\mathsf{int} \to \mathsf{int}.$
$\quad \mathsf{fn}\ n.\ \mathsf{if}(n \langle\ 2)1(\mathsf{fact}(n-1) * n))\ \mathsf{in}$
$\mathsf{factorial}(5)$

This program has type int and evaluates to 120.

2 Type Equivalence

In many languages, there are multiple ways to write a single type. Thus, it becomes important to determine when type expressions τ_1 and τ_2 represent the same type. We write $\vdash \tau_1 \cong \tau_2$ when τ_1 and τ_2 are equivalent types. The typing rule that allows us to make use of equivalence is this:

$$\frac{\vdash \tau_1 \cong \tau_2 \quad \Gamma \vdash e\!:\!\tau_1}{\Gamma \vdash e\!:\!\tau_2}$$

This means that if τ_1 and τ_2 are equivalent types, if e has type τ_1 then it also has type τ_2 in the same context.

For an example of equivalent types, suppose we extend tF with $(\tau_1 * \tau_2) \to \tau_3 \cong \tau_1 \to (\tau_2 \to \tau_3)$. Recall that curried functions are applicable to pairs and uncurried functions are applicable to single expressions. $(\tau_1 * \tau_2) \to \tau_3 \cong \tau_1 \to (\tau_2 \to \tau_3)$, because we can write down a (compiler-inserted) bijection between the types for *curry* and *uncurry* as follows:

$$\mathcal{C}[\![e\!:\!(\tau_1 * \tau_2) \to \tau_3]\!] = \mathcal{C}[\![\lambda x\!:\!\tau_1.\lambda y\!:\!\tau_2.e\langle x, y\rangle\!:\!\tau_1 \to (\tau_2 \to \tau_3)]\!]$$
$$\mathcal{C}[\![e\!:\!\tau_1 \to (\tau_2 \to \tau_3)]\!] = \mathcal{C}[\![\lambda p\!:\!\tau_1 * \tau_2.e(\pi_1 p)(\pi_2 p)\!:\!(\tau_1 * \tau_2) \to \tau_3]\!]$$

2.1 Name vs. Structural Equivalence

When trying to decide when two types are equivalent, we must also consider name equivalence and structural equivalence. These two types of equivalence are language-dependent. For example, in tF, two types are equivalent if they have identical syntactic form. This is a restricted form of structural equivalence. However, in C, Java, and Pascal, the name of a particular data structure is part of its type identity. For example, in C:

$$\mathsf{struct\ foo}\{\mathsf{int\ a, b;}\} \ncong \mathsf{struct\ bar}\{\mathsf{int\ a, b;}\}$$

since the name foo $\neq$ the name bar. However, in C, it is possible to get structural equivalence using the typedef command as follows:

typedef struct foo Foo;

In this case, Foo is declared to be another name for the type struct foo, and Foo $\cong$ struct foo. In Modula-3, one can bind a type to a name to brand it with a unique identity. Thus, name equivalence is achieved through explicit branding. To see how this works, consider the following example:

$$\mathsf{TYPE\ intpair} = \mathsf{RECORD\ x, y\!:\!int\ END}$$

$$\mathsf{TYPE\ foo} = \mathsf{BRANDED\ intpair}, \mathsf{bar} = \mathsf{BRANDED\ intpair};$$

Then foo $\ncong$ bar $\ncong$ intpair, but intpair $\cong$ RECORD x, y : int END via structural equivalence.

3 Data Structures

Now that we understand type equivalence in tF, we would like to explore the possibility of adding data structures to the langugae. For example, in C, we can create a binary tree as follows:

```
struct Tree{
    bool leaf;
    union{
        struct{Tree *left, Tree *right; } children;
        int value;
    }u;
}
```

How might we express this in our type notation? Here's an attempt:

$$\tau = \mathsf{bool} * (\tau * \tau + \mathsf{int}).$$

Unfortunately this is an equation and cannot be used as it is. Instead we need the solution to the above equation.

Another example of an expression that we cannot assign a type to in this langugae is $(\lambda\ x\ (x\ x))$. In lambda calculus, we can write down the following terms:

$$(\lambda\ x\ (x\ x))(\lambda\ x\ (x\ x))\ \text{(diverges)}$$

$$(\lambda\ x\ (x\ x))(\lambda\ y\ \#f) \Downarrow \#f.$$

The first term diverges and can be assigned a type τ that solves the equation $\tau = \tau \rightarrow \tau_1$ for any τ_1. The second terms does not diverge, but in order to assign a type to the above expression, we need the solution to the equation $\tau = \tau \rightarrow \mathsf{bool}$.

4 Fixed Point Type Constructor

It is clear from these examples that in order to make tF more useful to us, we need to be able to solve equations of the form $X = \tau$ where X is a type variable mentioned in the type expression τ. In order to solve these equations, we assume that we have a type constructor $\mu X.\tau$ which produces the solution to the above equation. This is an analogue of rec $x\ e$ for types and is similar to an ML datatype declaration.
Using this new type constructor, we can define many useful types such as:
$\mathsf{tree} \triangleq \mu T.T * T + \mathsf{int}$ (binary tree)
$\mathsf{nat} \triangleq \mu N.\mathsf{unit} + N$
$0 \triangleq \mathsf{inl}(\#u)\!:\!\mathsf{nat}$
$1 \triangleq \mathsf{inr}(\mathsf{inl}(\#u))\!:\!\mathsf{nat}$
$2 \triangleq \mathsf{inr}(\mathsf{inr}(\mathsf{inl}(\#u)))\!:\!\mathsf{nat}$

$\vdots$
$\mathsf{successor} \triangleq \lambda n{:}\mathsf{nat}.\mathsf{inr}(n)$ (increment)

In order to fully understand types, we must also study open and closed recursion. In many modern languages, types are allowed to refer to one another arbitrarily (even if they are in different source files)! Open recursion occurs when the type expression is not closed. (The opposite of this is, of course, called closed recursion). Open recursion requires taking a fixed point over all types in scope.

For example, consider the following class definitions for Node and Edge:

```
class Node{
    Edge[ ] outgoing_edges;
    Edge[ ] incoming_edges;
}

class Edge{
    Node from;
    Node to;
}
```

Notice that Node refers to Edge and vice versa. So, we must take a fixed point when assigning their types. Thus, we have

$$\mathsf{Node} = \mu\mathsf{N}.\mathsf{array}[\mathsf{N} * \mathsf{N}] * \mathsf{array}[\mathsf{N} * \mathsf{N}]$$

and

$$\mathsf{Edge} = \mu\mathsf{E}.(\mathsf{array}[\mathsf{E}] * \mathsf{array}[\mathsf{E}]) * (\mathsf{array}[\mathsf{E}] * \mathsf{array}[\mathsf{E}]).$$

5 Revisiting Type Equivalence - (fold/unfold)

Unfortunately, by taking fixed points over all types in scope, we have created a problem for the type equivalence of tF. The difficulty is that we no longer have one unique syntactic form for a given type.
Now if we suppose that $\mu X.\tau$ solves $X = \tau$, we can substitute $\mu X.\tau$ for X wherever it appears in τ.
So, for the natural numbers, we get $\mu\mathsf{N}.(\mathsf{unit} + \mathsf{N}) = \mu\mathsf{N}.(\mathsf{unit} + (\mu\mathsf{N}.(\mathsf{unit} + \mathsf{N}))$ and $\mathsf{nat} = (\mathsf{unit} + \mathsf{nat}) = (\mathsf{unit} + (\mathsf{unit} + \mathsf{nat}))\ldots$
In order to develop a new formal definition of type equivalence, we will define the *unfolding* of type $\mu X.\tau$ to be $\tau\{\mu X.\tau/X\}$. And, we will write $\mu X.\tau \cong \tau\{\mu X.\tau/X\}$ to mean that $\mu X.\tau$ and its unfolding are equivalent types. Implicitly, this gives us the following notion of equivalence: type expressions are equivalent if they are fully substitutable for each other. A weaker notion of equivalence is that the types are isomorphic and the expressions must be explicitly mapped between types.
It will be more straightforward for now if we consider a recursive type and its unfolding to be isomosphic and explicitly shift values between the two types. In order to move between the recursive type $\mu X.\tau$ and the corresponding unfolding $\tau\{\mu X.\tau/X\}$, we define two new operators named fold and unfold:

$$\mathsf{fold}{:}\tau\{\mu X.\tau/X\} \to \mu X.\tau$$

$$\mathsf{unfold}{:}\mu X.\tau \to \tau\{\mu X.\tau/X\}.$$

Note that the fold operator packages the concrete value as the abstract value, while the unfold operator allows access to the internals of the value of the recursive type. In addition, there is a bijection between fold and unfold, and their types are isomorphic. In Winskel, (fold/unfold) are referred to as (abs/rep).
Next, we will give the typing and evaluation rules for the fold and unfold operators.
The typing rules are as follows:

$$\frac{\Gamma \vdash e{:}\tau\{\mu X.\tau/X\}}{\Gamma \vdash \mathsf{fold}_{\mu X.\tau} e{:}\mu X.\tau}$$

$$\frac{\Gamma \vdash e : \mu X.\tau}{\Gamma \vdash \textsf{unfold}\ e : \tau\{\mu X.\tau / X\}}$$

The evaluation rule is:

$$\textsf{unfold}(\textsf{fold}_{\mu X.\tau} e) \to e,$$

where the new evaluation contexts are given by

$$C ::= \ldots \mid \textsf{fold}\ C \mid \textsf{unfold}\ C.$$

Now that we have the typing and evaluation rules for the new operators fold and unfold, we would like to prove that $(\lambda\ x(x\ x))$ can be typed as $\mu T.T \to \textsf{bool}$.
In order to accomplish this, we first add the following declarations to explicitly show the types of fold and unfold as applied to the above expression:

$$\textsf{fold}_{\mu T.T \to \textsf{bool}}(\lambda x : (\mu T.T \to \textsf{bool}).((\textsf{unfold}\ x)\ x)).$$

Now we can prove that $(\lambda\ x\ (x\ x))$ can be typed as $\mu T.T \to \textsf{bool}$ making use of our typing rules. Here is the proof tree:

$$\cfrac{\cfrac{\cfrac{\cfrac{\{x : \mu T.T \to \textsf{bool}\} \vdash x : (\mu T.T \to \textsf{bool})}{\{x : \mu T.T \to \textsf{bool}\} \vdash \textsf{unfold}\ x : (\mu T.T \to \textsf{bool}) \to \textsf{bool} \ldots}}{\{x : \mu T.T \to \textsf{bool}\} \vdash ((\textsf{unfold}\ x)\ x) : \textsf{bool}}}{\vdash [\lambda x : (\mu T.T \to \textsf{bool}).((\textsf{unfold}\ x)\ x)] : (\mu T.T \to \textsf{bool}) \to \textsf{bool}}}{\vdash [\textsf{fold}_{\mu T.T \to \textsf{bool}}(\lambda x : (\mu T.T \to \textsf{bool}).((\textsf{unfold}\ x)\ x))] : \mu T.T \to \textsf{bool}}$$

The fold and unfold operators are used somewhat differently in real languages. In Java and Modula-3, recursive types and their unfoldings are substitutable. In these languages, the fold and unfold operators are automatically supplied whenever needed. In ML, the datatype constructor is fold. And, the match operation provides an implicit unfold for each arm of the sum. In CLU and C, explicit use of operators is required to shift between the abstract and concrete values. CLU uses up and down operators to replace fold and unfold. In each of these languages there is no "pure" recursive type constructor; recursive types are constructed using type constructors with other functionality. For example, in ML, datatypes are used to introduce recursion but are also sum types.

6 Capturing the Untyped λ

Observe that all λ calculus expressions can be used in any context, and that evaluation never gets stuck. Thus, we would like to figure out how to capture the untyped λ in a typed language. Lambda calculus terms have the following type: $\Lambda = \mu X.X \to X$. Note that this is isomorphic to $(\mu X.X \to X) \to (\mu X.X \to X)$, i.e.,

$$\Lambda = \mu X.X \to X \cong (\mu X.X \to X) \to (\mu X.X \to X)$$

So $\Lambda \cong \Lambda \to \Lambda$.
We can develop a translation $\mathcal{D}$ from λ to $\lambda^{\to\mu}$. We will do this through using the fold and unfold operators as follows:

- $\mathcal{D}[\![x]\!] = x$
- $\mathcal{D}[\![\lambda\ x\ e]\!] = (\textsf{fold}_{\mu X.X \to X}(\lambda x : \Lambda.\mathcal{D}[\![e]\!]))$
- $\mathcal{D}[\![e_1\ e_2]\!] = (\textsf{unfold}\ \mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!]$

Now that we have figured out how to translate from $\lambda \to \lambda^{\to\mu}$, we can use recursive types to write the Y_τ operator as an ordinary expression. We can then desugar $\textsf{rec} f : \tau.e_r$ as $Y_\tau(\lambda f : \tau.e_r)$ and dispense with the built-in operator rec.

7 Constructing a Model

Currently our semantics model τ as having the domain $\mathcal{T}[\![\tau]\!]$. We must determine how to model the domain for the new type constructor, $\mu X.\tau$, i.e., we must determine how to model $\mathcal{T}[\![\mu X.\tau]\!]$. Earlier in the lecture, we noted that $\mu X.\tau \cong \tau\{\mu X.\tau/X\}$. Thus, we expect isomorphism to hold in their domains as well. Thus, we hope that

$$\mathcal{T}[\![\mu X.\tau]\!] \cong \mathcal{T}[\![\tau\{\mu X.\tau/X\}]\!].$$

For an example of this, consider the natural numbers. Let $\mathsf{N} = \mathcal{T}[\![\mu \mathsf{N}.\mathsf{unit} + \mathsf{N}]\!]$. We wish to require that $\mathcal{T}[\![\mu \mathsf{N}.\mathsf{unit}\ \mathsf{N}]\!] \cong \mathcal{T}[\![\mathsf{unit} + (\mu \mathsf{N}.\mathsf{unit} + \mathsf{N})]\!]$, i.e., that $\mathsf{N} \cong \mathsf{unit} + \mathsf{N}$. Modeling these types require solutions to domain equations that we have been using all along.
In order to solve the domain equations, we assume that a constructor for recursive domains exists and is given by $\mu D.\mathcal{F}(D)$, where the functor $\mathcal{F}$ maps one domain into another domain. If $D = \mu X.\mathcal{F}(X)$, then $\mathcal{F}(D)$ produces a domain related to D by continuous functions up and down that are inverses of one another.

We define up and down as follows:

$$\mathsf{down} : D \to \mathcal{F}(D)$$

$$\mathsf{up} : \mathcal{F}(D) \to D,$$

where $D \cong \mathcal{F}(D)$.

Because up and down are continuous,

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots \in D \Rightarrow \mathsf{up}(\bigsqcup d_i) = \bigsqcup \mathsf{up}(d_i)$$

$$e_0 \sqsubseteq e_1 \sqsubseteq e_2 \sqsubseteq \ldots \in \mathcal{F}(D) \Rightarrow \mathsf{down}(\bigsqcup e_i) = \bigsqcup \mathsf{down}(e_i).$$

Recursive types also introduce names for types. Thus we need a type environment $\chi : \mathsf{Type} \to \mathsf{Domain}$ which we define inductively. In our definition, $\mathcal{T}[\![\tau]\!]\chi$ gives the domain corresponding to type τ, given type variables whose meaning is defined by χ.

Here are the rules for evaluating $\mathcal{T}[\![\tau]\!]\chi$ inductively:

- $\mathcal{T}[\![\mathsf{unit}]\!]\chi = U$
- $\mathcal{T}[\![\mathsf{int}]\!]\chi = Z$
- $\mathcal{T}[\![X]\!]\chi = \chi(X)$
- $\mathcal{T}[\![\tau_1 * \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \times \mathcal{T}[\![\tau_2]\!]\chi$
- $\mathcal{T}[\![\tau_1 \to \tau_2]\!]\chi = \mathcal{T}[\![\tau_1]\!]\chi \to (\mathcal{T}[\![\tau_2]\!]\chi)_\perp$
- $\mathcal{T}[\![\mu X.\tau]\!]\ X = \mu D.\mathcal{T}[\![\tau]\!]\chi[X \to D]$