CS411 Notes 15 Recursive Types I

A. Demers

9 Apr 2001

1 Motivation

Our little language has grown fairly rich. It includes records, union types, higher-order functions as first-class entities, subtyping, etc. But there remains a fundamental missing capability — recursion in types; that is, the ability for a type signature to refer to itself.

A simple example illustrates how essential this feature is. Suppose we want to define a type L of list structures similar to Lisp S-expressions. A first attempt might look something like this:

$$\textbf{rectype}\ L\ \sim\ \textbf{sum}(\text{nil} : \textbf{prod}(), \text{cons} : \textbf{prod}(\text{car} : L, \text{cdr} : L))$$

This declaration is isomorphic to what you'd write in just about any modern programming language. We can't write it in the language fragment we've developed up to now, because we have no way to introduce a type name L and then refer to it from within the type expression defining L. This recursive use of a new type name is absolutely essential here. A critical defining property of a "list" is that it can contain substructures that are also lists. Without the ability to express this, our language is fundamentally crippled.

Recall that any attempt to eliminate recursion from a function definition by "in-line expansion" resulted in an intractible infinite function body. To give an operational semantics for recursive functions, we simulated the in-line expansion incrementally, as substitutions associated with evaluation rules for function application.

Similarly, any attempt to eliminate recursion from a type definition for lists by in-line expansion would generate infinite type expressions. So we take a similar approach — we unwind recursive type definitions incrementally, perform

one-level substitutions as required in rules for "abstraction" constructors and "representation" selectors.

The process is a bit delicate. We'll be extending the type expressions in a fundamental way: adding type names, the possibility of free type variables and capture, and in general making it nontrivial to guarantee "well-formedness" or "consistency" of type assignments, configurations, etc.

All that said, the details are surprisingly simple.

2 Syntax

Type Variables Recursive type declarations will require *type names*, a new syntactic set. We use

$$X, X_1, X_2, Y, Z, \ldots$$

for type names.

Types

$$\tau \ ::= \ X \mid (\mathbf{mu}\ X\ \mathbf{dot}\ \tau)$$

The first new form of type expression is just a reference to a type name. A **mu** type expression denotes a recursive type. The name X may appear free in the body τ of the **mu** expression; intuitively, it is bound to the type being defined.

Expressions

$$e \ ::= \ \mathbf{abs}_\tau(e) \mid \mathbf{rep}_\tau(e)$$

These are new expression forms for dealing with values of recursive type. The $\mathbf{abs}_\tau$ and $\mathbf{rep}_\tau$ constructs are used to move between the "abstract" recursive type and its "representation" — basically, to control the contexts in which is is possible to examine the internal structure of a value. Some examples below should clarify this. In an expression of either kind, the subscript/annotation τ must be a **mu** type expression. This is enforced by the typing rules below.

Values

$$v \ ::= \ \mathbf{abs}_{\tau}(v)$$

As with **sum** types, a value is produced by applying a constructor (in this case **abs**) to a value.

3 Typing Rules

There are two typing rules for recursive types. They illustrate the intimate connection between the type

$$\mathbf{mu}\ X\ \mathbf{dot}\ \tau$$

which can be thought of as an "abstract" recursive type, and its "unrolled" version

$$[\mathbf{mu}\ X\ \mathbf{dot}\ \tau/X]\tau$$

which can said to reveal the type's "representation." The **abs** and **rep** constructs convert between these two views of a recursive type.

$$\frac{\pi \vdash e : [\mathbf{mu}\ X\ \mathbf{dot}\ \tau/X]\tau}{\pi \vdash \mathbf{abs}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(e) : \mathbf{mu}\ X\ \mathbf{dot}\ \tau} \qquad \text{(T15.1)}$$

$$\frac{\pi \vdash e : \mathbf{mu}\ X\ \mathbf{dot}\ \tau}{\pi \vdash \mathbf{rep}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(e) : [\mathbf{mu}\ X\ \mathbf{dot}\ \tau/X]\tau} \qquad \text{(T15.2)}$$

We are actually skating on rather thin ice here. Since we have introduced type names, there are issues of free names in type expressions, capture of type names when substitutions are performed by the above rules, equivalence of recursive types, etc. We shall deal with these issues later. For now, just keep in mind that there are unanswered questions we'll need to address.

4 Evaluation Rules

We introduced only two new expression constructs — **abs** and **rep** — and consequently we have two new evaluation rules.

$$\frac{\langle e, \phi, \sigma\rangle \rightarrow \langle v, \sigma'\rangle}{\langle \mathbf{abs}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(e), \phi, \sigma\rangle \rightarrow \langle \mathbf{abs}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(v), \sigma'\rangle} \qquad \text{(E15.3)}$$

A well typed **abs** expression (which by the first typing rule above must consist of an **abs** constructor applied to a well typed subexpression of the unwound representation type) is reduced to a value by reducing its subexpression to a value and then embedding the result in the **abs** constructor.

$$\frac{\langle e, \phi, \sigma\rangle \to \langle \mathbf{abs}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(v), \sigma'\rangle}{\langle \mathbf{rep}_{\mathbf{mu}\ X\ \mathbf{dot}\ \tau}(e), \phi, \sigma\rangle \to \langle v, \sigma'\rangle} \qquad \text{(E15.4)}$$

This rule says **rep** is a left inverse of **abs**. To compute with a value of a recursive type, apply **rep** to map it to a value of the representation type.

5 Example

Here is an extended example, using a recursive type to simulate the natural numbers. Let

$$NN \quad \equiv \quad \mathbf{mu}\ X\ \mathbf{dot}\ \mathbf{sum}(z : \mathbf{prod}(), s : X)$$

A natural number is either zero (the 'z' case) or the successor of another natural number (the 's' case). The representation as discussed above is the result of unrolling the recursive type definition by substitution:

$$\begin{aligned} NNR \quad &\equiv \quad [NN/X](\mathbf{sum}(z : \mathbf{prod}(), s : X) \\ &\quad = \quad \mathbf{sum}(z : \mathbf{prod}(), s : NN) \end{aligned}$$

The **abs** constructor can be thought of as mapping from NNR to NN; and the **rep** constructor can be thought of as mapping from NN to NNR

A few well typed expressions:

$$\begin{aligned} &\text{Zero} : NN \\ &\text{Zero} \ \sim \ \mathbf{abs}_{NN}(\mathbf{inj}_{NNR}(z \sim \langle\rangle)) \end{aligned}$$

To construct a value of a recursive type, it must be possible to construct a value of the represenation type without requiring another value of the recursive type — i.e., there must be a "base case." That is why most uses of recursive types involve **sum** types Here $\mathbf{inj}_{NNR}$ using the z variant has the required behavior.

$$\begin{aligned} &\text{One} : NN \\ &\text{One} \ \sim \ \mathbf{abs}_{NN}(\mathbf{inj}(s \sim \text{Zero}) \end{aligned}$$

This uses the other case of $\mathbf{inj}_{NNR}$ (the s case). This requires a preexisting NN value, but we already have Zero at hand.

$$\begin{array}{l}\text{Suc} : \mathbf{fun}(NN)NN \\ \text{Suc} \ \sim \ \mathbf{lambda}\ n : NN\ \mathbf{dotabs}_{NN}(\mathbf{inj}(s \sim n)))\end{array}$$

$$\begin{array}{l}\text{Pred} : \mathbf{fun}(NN)NN \\ \text{Pred} \ \sim \ \mathbf{lambda}\ n : NN\ \mathbf{dot}\ \mathbf{case}\ \mathbf{rep}_{NN}(n)(z \Rightarrow n,\ s \Rightarrow s)\end{array}$$

These are simple functions that can be defined without recursion. To get a more interesting function like addition we need recursion. As in some of our earlier examples, can "cheat" and elicit recursive behavior by assignment to a function valued variable:

$$\begin{array}{l}\text{Plus} : \mathbf{fun}(NN)\mathbf{fun}(NN)NN \\ \text{Plus} \ \sim \ \mathbf{let}\ p \ \sim \ \mathbf{newvar}(\ldots)\ \mathbf{in} \\ \qquad p \ \leftarrow \ \mathbf{lambda}\ x : NN\ \mathbf{dot}\ \mathbf{lambda}\ y : NN\ \mathbf{dot} \\ \qquad\qquad \mathbf{case}\ \mathbf{rep}_{NN}(x)(z \Rightarrow y,\ s \Rightarrow \text{Suc}(p(s, y)))\end{array}$$

Later we'll show how to get recursive behavior by defining a "least fixed point" operator as a function of a recursive type, without either assignment or explicit recursion in the language itself.

6 Preview

As we pointed out above, there are a few issues involving free type variables and capture that we have to address before we can argue that our type rules are sound. That's for the next set of notes ...