Lecture Notes on Recursive Types

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 13
October 7, 2003

In the last two lectures we have seen two critical concepts of programming languages: parametric polymorphism (modeled by universal types) and data abstraction (modeled by existential types). These provide quantification over types, but they do not allow us to define types recursively. Clearly, this is needed in a practical language. Common data structures such as lists or trees are defined inductively, which is a restricted case of general recursion in the definition of types [Ch. 19.3].

So far, we have considered how to add a particular recursive type, namely lists, to our language as a primitive by giving constructors (`nil` and `cons`), a discriminating destructor (`listcase`). For a realistic language, this approach is unsatisfactory because we would have to extend the language itself every time we needed a new data type. Instead we would like to have a uniform construct to define new recursive types as we need them. In ML, this is accomplished with the `datatype` construct. Here we use a somewhat lower-level primitive—we return to the question how this is related to ML at the end of this lecture.

As a first, simple *non-recursive* example, consider how we might implement a three-element type.

```
datatype Color = Red | Green | Blue;
```

Using the singleton type 1 (`unit`, in ML), we can define

$$
\begin{array}{lcl}
\mathit{Color} & = & 1 + (1 + 1) \\
\mathit{Red} : \mathit{Color} & = & \mathsf{inl}\,(\,) \\
\mathit{Green} : \mathit{Color} & = & \mathsf{inr}(\mathsf{inl}\,(\,)) \\
\mathit{Blue} : \mathit{Color} & = & \mathsf{inr}(\mathsf{inr}\,(\,)) \\
\mathit{ccase} & : & \forall s.\mathit{Color} \to (1 \to s) \to (1 \to s) \to (1 \to s) \to s \\
 & = & \Lambda s.\lambda c.\lambda y_1.\lambda y_2.\lambda y_3. \\
 & & \quad \mathsf{case}\ c \\
 & & \qquad \mathsf{of}\ \mathsf{inl}(c_1) \Rightarrow y_1\, c_1 \\
 & & \qquad \mid \mathsf{inr}(c_2) \Rightarrow \mathsf{case}\ c_2 \\
 & & \qquad\qquad\qquad \mathsf{of}\ \mathsf{inl}(z_2) \Rightarrow y_2\, z_2 \\
 & & \qquad\qquad\qquad \mid \mathsf{inr}(z_3) \Rightarrow y_3\, z_3
\end{array}
$$

Recall the notation $\lambda x.e$ for a non-recursive function and $\Lambda t.e$ for a type abstraction. The ccase constructs invokes one of its arguments y_1, y_2, or y_3, depending on whether the argument c represents red, green, or blue.

If we try to apply the technique, for example, to represent natural numbers as they would be given in ML by

```
datatype Nat = Zero | Succ of Nat;
```

we would have

$$\mathit{Nat} = 1 + (1 + (1 + \cdots))$$

where

$$n : \mathit{Nat} = \underbrace{\mathsf{inr}(\ldots(\mathsf{inr}(}_{n \text{ times}}\,\mathsf{inl}\,(\,))))$$

In order to make this definition recursive instead of infinitary we would write

$$\mathit{Nat} \simeq 1 + \mathit{Nat}$$

where we leave the mathematical status of $\simeq$ purposely vague, but one should read $\tau \simeq \sigma$ as "*τ is isomorphic to σ*". Just as with the recursion at the level of expressions, it is more convenient to write this out as an explicit definition using a recursion operator.

$$\mathit{Nat} = \mu t.1 + t$$

We can unwind a recursive type $\mu t.\sigma$ to $\{\mu t.\sigma/t\}\sigma$ to obtain an isomorphic type.

$$\mathit{Nat} = \mu t.1 + t \simeq \{\mu t.1 + t/t\}1 + t = 1 + \mu t.1 + t = 1 + \mathit{Nat}$$

In order to obtain a reasonable system for type-checking, we have constructors and destructors for recursive types. They can be considered "witnesses" for the unrolling of a recursive type.

$$\frac{\Gamma \vdash e : \{\mu t.\tau / t\}\tau \quad \Gamma \vdash \mu t.\tau \ \mathsf{type}}{\Gamma \vdash \mathsf{roll}(e) : \mu t.\tau} \qquad \frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \mathsf{unroll}(e) : \{\mu t.\tau / t\}\tau}$$

The operational semantics and values are straightforward; the difficulty of recursive types lies entirely in the complexity of the substitution that takes place during the unrolling of a recursive type.

$$\frac{e \mapsto e'}{\mathsf{roll}(e) \mapsto \mathsf{roll}(e')} \qquad \frac{v \ \ \mathsf{value}}{\mathsf{roll}(v) \ \ \mathsf{value}}$$

$$\frac{e \mapsto e'}{\mathsf{unroll}(e) \mapsto \mathsf{unroll}(e')} \qquad \frac{v \ \ \mathsf{value}}{\mathsf{unroll}(\mathsf{roll}(v)) \mapsto v}$$

Now we can go back to the definition of specific recursive types, using natural numbers built from zero and successor as the first example.

$$\begin{array}{lcl}
Nat & = & \mu t.1 + t \\
Zero : Nat & = & \mathsf{roll}(\mathsf{inl}\,(\,)) \\
Succ : Nat \to Nat & = & \lambda x.\mathsf{roll}(\mathsf{inr}\, x) \\
ncase & : & \forall s.Nat \to (1 \to s) \to (\mathtt{nat} \to s) \to s \\
 & = & \Lambda s.\lambda n.\lambda y_1.\lambda y_2. \\
 & & \quad \mathsf{case\ unroll}(n) \\
 & & \quad\quad \mathsf{of\ inl}(z_1) \Rightarrow y_1\, z_1 \\
 & & \quad\quad\ \ \mid \mathsf{inr}(z_2) \Rightarrow y_2\, z_2
\end{array}$$

In the definition of $ncase$ we see that $z_1 : 1$ and $z_2 : Nat$, so that y_2 is really applies to the predecessor of n, while y_1 is just applied to the unit element.

Polymorphic recursive types can be defined in a similar manner. As an example, we consider lists with elements of type r.

$$\begin{array}{lcl}
r\ List & = & \mu t.1 + r \times t \\
Nil & : & \forall r.r\ List \\
 & = & \Lambda r.\mathsf{roll}(\mathsf{inl}\,(\,)) \\
Cons & : & \forall r.r \times r\ List \to r\ List \\
 & = & \Lambda s.\lambda p.\mathsf{roll}(\mathsf{inr}\, p) \\
lcase & : & \forall s.\forall r.r\ List \to (1 \to s) \to (r \times r\ List \to s) \to s \\
 & = & \Lambda s.\Lambda r.\lambda l.\lambda y_1.\lambda y_2. \\
 & & \quad \mathsf{case\ unroll}(l) \\
 & & \quad\quad \mathsf{of\ inl}(z_1) \Rightarrow y_1\, z_1 \\
 & & \quad\quad\ \ \mid \mathsf{inr}(z_2) \Rightarrow y_2\, z_2
\end{array}$$

If we go back to the first example, it is easy to see that representation of data types does not quite match their use in ML. This is because we can see the complete implementation of the type, for example, $Color = 1 + (1 + 1)$. This leads to a loss of data abstraction and confusion between different data types. Consider another ML data type

```
Answer = Yes | No | Maybe;
```

This would also be represented by

$$Answer = 1 + (1 + 1)$$

which is the *same* as $Color$. Perhaps this does not seem problematic until we realize that `Yes` = `Red`! This is obviously meaningless and creates incompatibilities, for example, if we decide to change the order of definition of the elements of the data types.

Fortunately, we already have the tool of data abstraction to avoid this kind of confusion. We therefore combine *recursive types* with *existential types* to model the `datatype` construct of ML. Using the first example,

```
datatype Color = Red | Green | Blue;
```

we would represent this

$$\exists c.c \times c \times c \times \forall s.c \to (1 \to s) \to (1 \to s) \to (1 \to s) \to s$$

The implementation will have the form

$$\texttt{pack}(1 + (1 + 1), \texttt{pair}(\mathsf{inl}\,(\,), \texttt{pair}(\mathsf{inr}(\mathsf{inl}\,(\,)), \ldots)))$$

Upon opening an implementation of this type we can give its components the usual names. With this strategy, $Color$ and $Answer$ can no longer be confused.

We close this section with a curiosity regarding recursive types. We can use them to type a simple, non-terminating expression that does *not* employ recursive functions! The pure λ-calculus version of this function is $(\lambda x.x\,x)\,(\lambda x.x\,x)$. Our example is just slight more complicated because of the need to roll and unroll recursive types.

We define

$$\begin{array}{lcl}
\omega & = & \mu t.t \to t \\
x{:}\omega & \vdash & \mathsf{unroll}(x) : \omega \to \omega \\
x{:}\omega & \vdash & \mathsf{unroll}(x)\,x : \omega \\
\cdot & \vdash & \lambda x.\mathsf{unroll}(x)\,x : \omega \to \omega \\
\cdot & \vdash & \mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x) : \omega \\
\cdot & \vdash & (\lambda x.\mathsf{unroll}(x)\,x)\,\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x) : \omega
\end{array}$$

When we execute this term we obtain

$$\begin{array}{l}
(\lambda x.\mathsf{unroll}(x)\,x)\,\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x) \\
\mapsto \mathsf{unroll}(\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x))\,(\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x)) \\
\mapsto (\lambda x.\mathsf{unroll}(x)\,x)\,(\mathsf{roll}(\lambda x.\mathsf{unroll}(x)\,x))
\end{array}$$

so it reduces to itself in two steps.

While we will probably not prove this in this course, recursive types are necessary for such an example. For any other (pure) type construct we have introduced so far, all functions are terminating if we do not use recursion at the level of expressions.