15-312 Foundations of Programming Languages
Recitation 7: Recursive Types

Daniel Spoonhower
spoons+@cs

October 8, 2003

1 Another Recursive Type

First, we'll consider another datatype with which you have some experience.

```
datatype exp =
     Num of int
   | Plus of exp * exp
```

As should be obvious, this datatype represents expressions of a simple arithmetic language. How would we write the type for this datatype (as we discussed yesterday in lecture)? We could start with a sum:

$$\texttt{int} + (\texttt{exp} * \texttt{exp}) \ ?$$

Of course `exp` is the type that we are trying to define! So we need to use a recursive type and replace our uses of `exp` with instances of the recursive (type) variable.

$$\texttt{exp} = \mu t.\texttt{int} + (t \times t)$$

What are some examples of values of this datatype (as we'd write them in SML)? Consider `Num(5)`, `Plus(Num(3),Num(4))`, and

`Plus(Plus(Num(2),Num(4)),Num(7))`

How would we write these values using our recursive type? A first attempt might look like `inl(5)`. (Why is this not correct?) Here are two examples of what we are really looking for:

```
roll(inl(5))    roll(inr(pair(roll(inl(3)),roll(inr(4)))))
```

(Can you give the translation of the last example?) What are the constructors for `exp`? (What are their types?) Think about it before you turn the page.

$$
\begin{array}{lcl}
\texttt{Num} & : & \texttt{int} \rightarrow \texttt{exp} \\
 & = & \texttt{fn}\,\texttt{n} : \texttt{int => roll(inl(n))} \\
\texttt{Plus} & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{exp} \\
 & = & \texttt{fn}\,\texttt{x} : \texttt{exp => fn}\,\texttt{y} : \texttt{exp => roll(inr(pair(x,y)))}
\end{array}
$$

What about the destructor? Give its type and implementation.

2 More On Datatypes

Using sum types, existential types and parametric polymorphism, we can also build a τ `option`, just as it appears in Standard ML.

```
datatype 'a option = NONE | SOME of 'a
```

What type would we give to this datatype? Perhaps something like,

$$\forall t.1 + t$$

We might also write this as $\forall t.\mu u.t + 1$, but the u is unused, and (as you will remember from lecture) the implementation of the datatype is hidden from the user anyway; we'll see more on this in a moment. What are the types and implementations for the constructors and the `case` function?

As we've just mentioned, SML datatypes are abstract: they hide their implementation from users. How might we use an existential type to hide our implementation?

$$\texttt{option} = \forall t.\exists u.u \times (t \rightarrow u) \times \forall s.u \rightarrow (1 \rightarrow s) \rightarrow (t \rightarrow s) \rightarrow s$$

Notice that the second constructor and the `case` naturally share the type parameter t. An implementation of this datatype might look something like:

$$\texttt{Fn t => pack}(1 + t, \texttt{pair(inl(()),pair(fn x:t => inr(x),}\ldots\texttt{))}$$

3 More on Recursion

In lecture yesterday, we saw the type ω, the type of function which can be applied to itself. Recall,

$$\omega = \mu t.t \rightarrow t$$
$$(\texttt{roll(fn}\,\texttt{x} : \omega \texttt{ => unroll(x)}\ x\texttt{)}) : \omega$$

We might read the type ω (in unrolled form) as "given a function that might be applied to itself, return a function that might be applied to itself." While ω is certainly a curiosity, we can do something more useful with one of its relatives:

$$\forall s.\mu t.t \rightarrow s$$

Below, we will consider one particular instance of this type,

$$\mu t.t \rightarrow \texttt{int} \rightarrow \texttt{int}$$

Before we continue, remember our implementation of the factorial function:

```
rec fact : int -> int =>
  fn x : int =>
    if x = 0 then 1
    else x * fact (x - 1)
    fi
```

The `rec` construct allows us to make an assumption (in the body of the expression) about the existence of a function from `int` to `int`. (Then after we've typechecked the body, we confirm that it really has the type we assumed.)

Let's make a similar assumption. Let `f` be a variable of type

$$\mu t.t \rightarrow \texttt{int} \rightarrow \texttt{int}$$

We'd like `f` to stand for a "recursively defined function from `int` to `int`," but we can't apply `f` as it stands. Instead, we must first unroll it:

$$\texttt{unroll(f)} : t \rightarrow \texttt{int} \rightarrow \texttt{int}$$

We can apply the unrolled version, but only to expressions of our recursive type, for example `f`.

$$(\texttt{unroll(f) f}) : \texttt{int} \rightarrow \texttt{int}$$

Excellent! We are almost ready to write our `rec`-less version of factorial. First, we must recognize that `f` must be bound somewhere; we add an additional function abstraction to pass it in.

```
fn f : μt. t -> int -> int =>
  fn x : int =>
    if x = 0 then 1
    else x * (unroll(f) f) (x - 1)
    fi
```

(What's the type of this expression?) Finally, we'd like to write a function of type `int` $\rightarrow$ `int` (rather than expose users to all of this roll/unroll syntax). How do we form such an expression? Call the above expression `F`. Then we can write:

```
let fact = F roll(F) in
  fact 5
end
```

(Verify to yourself that this indeed is the expression we want.)

Next Week

Next week's recitation will be a review for the midterm. Being any questions you have about the material we've covered so far.