1 Another Recursive Type

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

```plaintext
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:

```
int + (exp * 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.

```
exp = \mu t. 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.
Num : int → exp
  = fnn : int => roll(inl(n))

Plus : exp → exp → exp
  = fnx : exp => fny : exp => roll(inr(pair(x,y)))

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 \( \tau \) option, just as it appears in Standard ML.

\[
\text{datatype } \tau \text{ option} = \text{NONE} | \text{SOME of } \tau
\]

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

\[
\forall \tau.1 + \tau
\]

We might also write this as \( \forall \tau.\mu u.\tau + 1 \), 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?

\[
\text{option} = \forall \tau.\exists u.\tau \times (\tau \to u) \times \forall u.\tau \to (1 \to s) \to (\tau \to s) \to s
\]

Notice that the second constructor and the case naturally share the type parameter \( \tau \). An implementation of this datatype might look something like:

\[
\text{Fn } \tau \mapsto \text{pack(1+\tau,\text{pair(inl(()),pair(fn } x:\tau \mapsto \text{inr(x),...))})}
\]

3 More on Recursion

In lecture yesterday, we saw the type \( \omega \), the type of function which can be applied to itself. Recall,

\[
\omega = \mu t. t \to t
\]

\[
(\text{roll(fnx : } \omega \mapsto \text{unroll(x) } x)) : \omega
\]

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

\[
\forall s, \mu t. t \to s
\]
Below, we will consider one particular instance of this type,

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

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

```haskell
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

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

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:

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

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

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

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.

```haskell
fn f : \mu t. 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:

```haskell
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.