Chapter 27

Typing Data

27.1 Recursive Types

27.1.1 Declaring Recursive Types

We saw in the previous lecture how `rec` was necessary to write recursive *programs*. But what about defining recursive *types*? Recursive types are fundamental to computer science: even basic data structures like lists and trees are recursive (since the rest of a list is also a list, and each sub-tree is itself a tree).

Suppose we try to type the program

```
{rec {length : ???
       {fun {l : ???} : number
         {if {empty? l}
             0
             {+ 1 {length {rest l}}}}}}
  {length {numCons 1 {numCons 2 {numCons 3 numEmpty}}}}}
```

What should we write in place of the question marks?

Let's consider the type of `l`. What kind of value can be an argument to `l`? Clearly a numeric cons, because that's the argument supplied in the first invocation of `length`. But eventually, a numeric empty is passed to `l` also. This means `l` needs to have *two* types: (numeric) cons and empty.

In languages like ML (and Java), procedures do not consume arguments of more than one distinct type. Instead, they force programmers to define a new type that encompasses all the possible arguments. This is precisely what a datatype definition, of the kind we have been writing in Scheme, permits us to do. So let's try to write down such a datatype in a hypothetical extension to our (typed) implemented language:

```
{datatype numList
    {[numEmpty]
     [numCons {fst : number}
              {rst : ???}]}
  {rec {length : (numList -> number) ...}
    {length ...}}}
```

We assume that a datatype declaration introduces a collection of *variants*, followed by an actual body that uses the datatype. What type annotation should we place on `rst`? This should be precisely the new type we are introducing, namely `numList`.

A datatype declaration therefore enables us to do a few distinct things all in one notation:

1. Give names to new types.
2. Introduce conditionally-defined types (*variants*).
3. Permit recursive definitions.

If these are truly distinct, we should consider whether there are more primitive operators that we may provide so a programmer can mix-and-match them as necessary.[1]

But how distinct are these three operations, really? Giving a type a new name would be only so useful if the type were simple (for instance, creating the name `bool` as an alias for `boolean` may be convenient, but it's certainly not conceptually significant), so this capability is most useful when the name is assigned to a complex type. Recursion needs a name to use for declaring self-references, so it depends on the ability to introduce a new name. Finally, well-founded recursion depends on having both recursive and non-recursive cases, meaning the recursive type must be defined as a collection of variants (of which at least one is not self-referential). So the three capabilities coalesce very nicely.

As you may have noticed above, the datatypes we have introduced in our typed language are a bit different from those we're using in Scheme. Our Scheme datatypes are defined at the top-level, while those in the implemented language enclose the expressions that refer to them. This is primarily to make it easier to deal with the scope of the introduced types. Obviously, a full-fledged language (like ML and Haskell) permits apparently top-level datatype declarations, but we'll make this simplifying assumption here.

27.1.2 Judgments for Recursive Types

Let's consider another example of a recursive type: a family tree.

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  ...}
```

This data definition allows us to describe as much of the genealogy as we know, and terminate the construction when we reach an unknown person. What type declarations ensue from this definition?

$$\mathit{unknown} : \rightarrow \mathit{FamilyTree}$$

$$\mathit{person} : \texttt{string} \times \mathit{FamilyTree} \times \mathit{FamilyTree} \rightarrow \mathit{FamilyTree}$$

[1] "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary".

This doesn't yet give us a way of distinguishing between the two variants, and of selecting the fields in each variant. In Scheme, we use **type-case** to perform both of these operations. A corresponding case dispatcher for the above datatype might look like

```
{FamilyTree-cases v
  [{unknown} ...]
  [{person n m f} ...]}
```

Its pieces would be typed as follows:

$$\frac{\Gamma \vdash v : \mathit{FamilyTree} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma[n \leftarrow \texttt{string}, m \leftarrow \mathit{FamilyTree}, f \leftarrow \mathit{FamilyTree}] \vdash e_2 : \tau}{\Gamma \vdash \{\mathit{FamilyTree\text{-}cases}\ v\ \{[\mathit{unknown}]\ e_1\}\ \{[\mathit{person}\ n\ m\ f]\ e_2\}\} : \tau}$$

In other words, to determine the type of the entire `FamilyTree-cases` expression, τ, we first ensure that the value being dispatched is of the right type. Then we must make sure each branch of the switch returns a τ.[2] We can ensure that by checking each of the bodies in the right type environment. Because `unknown` has no fields, its `cases` branch binds no variables, so we check e_1 in Γ. In the branch for `person`, however, we bind three variables, so we must check the type of e_2 in a suitably extended Γ.

Though the judgment above is for a very specific type declaration, the general principle should be clear from what we've written. Effectively, the type checker introduces a new type rule for each typed `cases` statement based on the type declaration at the time it sees the declaration. Writing the judgment above in terms of subscripted parameters is tedious but easy.

Given the type rules above, consider the following program:

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {person "Mitochondrial Eve" {unknown} {unknown}}}
```

What is the type of the expression in the body of the datatype declaration? It's *FamilyTree*. But when the value escapes from the body of the declaration, how can we access it any longer? (We assume that the type checker renames types consistently, so *FamilyTree* in one scope is different from *FamilyTree* in another scope—just because the names are the same, the types should not conflate.) It basically becomes an *opaque type* that is no longer usable. This does not appear to be very useful at all![3]

At any rate, the type checker permitted a program that is quite useless, and we might want to prevent this. Therefore, we could place the restriction that the type defined in the datatype (in this case, *FamilyTree*) should be different from the type of the expression body τ. This prevents programmers from inadvertently returning values that nobody else can use.

[2] Based on the preceding discussion, if the two cases needed to return different types of values, how would you address this need in a language that enforced the type judgment above?

[3] Actually, you could use this to define the essence of a module or object system. These are called *existential types*. But we won't study them further in this course.

Obviously, this restriction doesn't reach far enough. Returning a vector of *FamilyTree* values avoids the restriction above, but the effect is the same: no part of the program outside the scope of the datatype can use these values. So we may want a more stringent restriction: the type being different *should not appear free* in τ.

This restriction may be overreaching, however. For instance, a programmer might define a new type, and return a package (a vector, say) consisting of two values: an instance of the new type, and a procedure that accesses the instances. For instance,

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {with {unknown-person : FamilyTree {unknown}}
    {vector
      {person "Mitochondrial Eve"
              unknown-person
              unknown-person}
      {fun {v : FamilyTree} : string
        {FamilyTree-cases v
          [{unknown}      {error ...}]
          [{person n m f} n]}}}}}
```

In this vector, the first value is an instance of *FamilyTree*, while the second value is a procedure of type

$$\textit{FamilyTree} \rightarrow \texttt{string}$$

Other values, such as `unknown-person`, are safely hidden from access. If we lift the restriction of the previous paragraph, this becomes a legal pair of values to return from an expression. Notice that the pair in effect forms an *object*: you can't look into it, so the only way to access it is with the "public" procedure. Indeed, this kind of type definition sees use in defining object systems.

That said, we still don't have a clear description of what restriction to affix on the type judgment for datatypes. Modern programming languages address this quandary by affixing no restriction at all. Instead, they effectively force all type declarations to be at the "top" level. Consequently, no type name is ever unbound, so the issues of this section do not arise. When we do need to restrict access, we employ module systems to delimit the scope of type bindings.

27.1.3 Space for Datatype Variant Tags

One of the benefits programmers incur from using datatypes—beyond the error checking—is slightly better space consumption. (Note: "better space consumption" = "using less space".) Whereas without static type checking we would need tags that indicate both the type *and* the variant, we now need to store *only* the variant. Why? Because the type checker statically ensures that we won't pass the wrong kind of value to procedures! Therefore, the run-time system needs to use only as many bits as are necessary to distinguish

between all the *variants* of a type, rather than all the datatypes as well (in addition). Since the number of variants is usually quite small, of the order of 3–4, the number of bits necessary for the tags is usually small also.

We are now taking a big risk, however. In the liberal tagging regime, where we use both type and variant tags, we can be sure a program will never execute on the wrong kind of data. But if we switch to a more liberal tagging regime—one that doesn't store type tags also—we run a huge risk. If we perform an operation on a value of the wrong type, we may completely destroy our data. For instance, suppose we can somehow pass a *NumList* to a procedure expecting a *FamilyTree*. If the `FamilyTree-cases` operation looks only at the variant bits, it could end up accessing a `numCons` as if it were a `person`. But a `numCons` has only two fields; when the program accesses the third field of this variant, it is essentially getting junk values. Therefore, we have to be very careful performing these kinds of optimizations. How can we be sure they are safe?