Datatypes
Lecture Notes for cs173, Fall 2001

sk and rob

October 29, 2001

We’ve seen how to add lists to the language. There’s something unsatisfying, however, about having to pester the language designer for each new type we want. We therefore add a datatype mechanism to our language. This permits us to define new types in a program. The advantage of having the ability to add new types is, of course, that we can have complete control over a type’s layout, and that functions that are not supposed to know about the new type won’t accidentally accept them. In contrast, if we use lists to represent all types, we run the risk of applying completely irrelevant functions to some of our data. We’ve already seen that define-struct in Scheme avoids this risk; now we’re going to see how we can put a type-checked version of define-struct into the type system.

Let’s start with a concrete example. We’ll create a new type for lists, since we’ve already seen them as a special case. A list is either empty or non-empty, and a non-empty list holds two values:

{datatype nlist
 {empty}
 {cons {first : num} {rest : nlist}}
 <expr>}

This defines a new type nlist. An nlist can be built using either the procedure empty (a slight shift from tradition—previously empty was just a constant—but one we make for consistency with other datatype declarations) or the procedure cons. The empty constructor consumes no values and returns an empty nlist. The cons constructor consumes two values and produces a non-empty nlist; the first value must be a num, and the second another nlist. Here are a few examples of nlist’s:

{empty}
{cons 1 {empty}}
{cons 1 {cons 2 {empty}}}

(Note that we’ve implicitly assumed that procedures can take zero or more arguments. This is pretty easy to implement in our interpreter, and extremely easy to denote in
type system, so we may as well assume this from now on.)

Rip's `datatype` construct, in addition to the constructors, automatically provides the selectors `first` and `rest`, which have the following contracts:

```
first : nlist -> num
rest : nlist -> nlist
```

We can assume, in addition, that the `datatype` mechanism automatically defines a `dispatcher`. The dispatcher for `nlist` consumes two functions (one for each `variant` of the datatype) and a value of the datatype, and selects the function corresponding to the right variant:

```
dispatch-nlist : nlist x ((() -> num) x (num x nlist -> num)) -> num
```

We’ve assumed the dispatcher always returns numbers. This is a bit silly, but we’ll see soon how we can lift this kind of type restriction. Here’s a sample use of the dispatcher:

```
{rec length : (nlist -> num)
 {proc {l : nlist} : num
  {dispatch-nlist l
   {proc {} 0}
   {proc {a : num d : nlist}
    {+ 1 {length d}}}))
  {length {cons 1 {cons 2 {cons 3 {empty}}}}}}

How do we write a type judgment for the `nlist` datatype?

```
Γ ⊢ empty : () -> nlist
Γ ⊢ cons : num x nlist -> nlist
Γ ⊢ first : nlist -> num
Γ ⊢ rest : nlist -> nlist
Γ ⊢ dispatch-nlist : ...
Γ ⊢ e : t
```

Now let’s consider another example:

```
{datatype ntree
 {empty}
 {node {value : num}
    {left : ntree}
    {right : ntree}}
<expr>}
```
It similarly binds in the `<expr>` the following constructors, selectors and dispatch procedure:

- `empty : () → ntree`
- `node : num × ntree × ntree → ntree`
- `value : ntree → num`
- `left : ntree → ntree`
- `right : ntree → ntree`
- `dispatch-tree : ntree × ((() → num) × (num × ntree × ntree → num) → num)`

So its typing rules must be:

\[
\begin{array}{c}
\Gamma \vdash \{\text{datatype } ntree \ldots \text{ e}:t \}
\end{array}
\]

In general, a datatype declaration has the form:

\[
\begin{array}{c}
\Gamma \vdash \{\text{datatype } T \f C_1 \{S_{11} : T_{11}\} \{S_{12} : T_{12}\} \ldots \{S_{1M} : T_{1M}\} \f C_2 \{S_{21} : T_{21}\} \{S_{22} : T_{22}\} \ldots \{S_{2N} : T_{2N}\} : t \\
\}
\end{array}
\]

We haven’t yet seen a complete Rip program that uses `datatype`. Let’s write one now:

```{datatype nlist
{empty}
{cons {first : num}
  {rest : nlist}}
empty}
What is the value of this expression? Since the expression inside the datatype term is \texttt{empty} (which has type \texttt{nilist}), the whole expression has type \texttt{nilist}. What sense does that make? How can anyone use a value of type \texttt{nilist} outside its declaration? It seems, therefore, that we should place a sensible restriction on Rip’s datatype: \( t \) (the type of the expression) should not equal \( T \) (the new type being created by the datatype expression).

Actually, it’s clear that we can reintroduce this problem with just a little extra work. Suppose we were to return not \texttt{empty} but the pair of \texttt{empty} with \texttt{empty}. Then \( t \) does not equal \( T \), but \( t \) (the expression’s type) certainly contains two instances of \( T \) (the new type), neither of which is useful outside the datatype declaration. This suggests that the right condition to impose is that \( T \) does not appear free anywhere in \( t \).

Actually, this isn’t good enough either! We can’t use \( T \) outside the datatype declaration because there’s no way of prying a value of that type open. Well, actually, there is—it’s the dispatcher. We could, for instance, return a pair consisting of the value and its dispatcher. Then a user outside the datatype expression could access the innards of the value. This suggests that our no free occurrences of \( T \) in \( t \) restriction is draconian, so we should lift it entirely.

But returning a value-dispatcher pair is still not a fully satisfactory solution. There are two problems. First, it’s impossible to create two datatypes that refer to one another (one of them must come first, and it can’t refer to the second one). This is obviously a problem for practical programming.

Second, exporting and using the dispatcher is actually a little trickier than it seems. Suppose there are two distinct expressions, each of which defines a new type called \texttt{nilist}. Presumably, even if these types have the same selector names, we don’t want one’s \texttt{nilist} being accessed by another. (Worse still, they may have \textit{different} selector names, in which case the system might crash horribly if the dispatcher from one instance of \texttt{nilist} tries to access the selectors of a value of the other instance.)

For these reasons, languages typically put all datatype declarations at the top level. That way, all expressions in a program can see the declarations, so there’s no problem returning values of a datatype from an expression, and so forth. This introduces problems of namespace control: what if two different programmers used the same name? We solve this problem with module systems. That’s a long and interesting topic, but not one we can tackle in this course.