1 Custom Datatypes

Suppose we want to write a function to tell us which TAs are at office hours on a particular day. This function might have the type

\[
\text{ohTA : day \to string}
\]

But how should we represent a day? One option would be to use integers, with 0 being Monday and 6 being Sunday, and just use an \texttt{REQUIRES} clause to make sure the input is valid:

\[
\text{type day = int}
\]

(* \text{ohTA : day \to string}
 * \texttt{REQUIRES: 0 \leq n \leq 6}
 * \texttt{ENSURES: ohTA n \Rightarrow a string representing the TA(s) holding office}
 * \text{hours on the n-th day of the week, with 0 = Monday}
 * *)

\[
\text{fun ohTA (0 : day) : string = "Pankaj"}
| \text{ ohTA (1 : day) : string = "Blair"}
| \text{ ohTA (2 : day) : string = "Chris"}
| ...
\]

\[
\text{type day = int}
\]

(* \text{ohTA : day \to string}
 * \texttt{REQUIRES: 0 \leq n \leq 6}
 * \texttt{ENSURES: ohTA n \Rightarrow a string representing the TA(s) holding office}
 * \text{hours on the n-th day of the week, with 0 = Monday}
 * *)

\[
\text{fun ohTA (0 : day) : string = "Pankaj"}
| \text{ ohTA (1 : day) : string = "Blair"}
| \text{ ohTA (2 : day) : string = "Chris"}
| ...
\]

*Tree material based on notes by Brandon Bohrer*
Of course, the match is going to be incomplete because we won’t have cases for numbers less than 0 or greater than 6. If we wanted the function to do something other than raise Match in these cases, we’d have to come up with an alternative solution, e.g.

type day = int

(* ohTA : day -> string
 * REQUIRES: 0 <= n <= 6
 * ENSURES: ohTA n ==> a string representing the TA(s) holding office
 * hours on the nth day of the week, with 0 = Monday
 *)
fun ohTA (0 : day) : string = "Pankaj"
 | ohTA (1 : day) : string = "Blair"
 | ohTA (2 : day) : string = "Chris"
 ... 
 | ohTA _ : string = "Unexpected input"

This might lead to confusion if students begin to believe that Unexpected input is a 15-150 TA holding office hours on the 8th day of the week.

Just like we saw with integers vs. natural numbers, it would be nice if, instead, the type system took care of this for us and made sure that an invalid input just couldn’t be passed in.

A day is either

Sunday or
Monday or
Tuesday or
Wednesday or
Thursday or
Friday or
Saturday or
Sunday

And that’s it!

Wouldn’t it be nice if the datatype agreed? Fortunately, SML lets you define your own datatypes, and we can represent exactly the above constraints in SML:

datatype day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

This establishes Monday, ..., Sunday as constructors. Constructors are ways of building a value of a particular datatype. Once we’ve introduced the type day, it is the case that
Monday : day
...
Sunday : day

We can also use the constructors for pattern matching!

(* ohTA : day -> string
  * REQUIRES: true
  * ENSURES: ohTA d ==> a string representing the TA(s) holding office
  * hours on day d
fun ohTA (Monday : day) : string = "Pankaj"
  | ohTA (Tuesday : day) : string = "Blair"
  | ohTA (Wednesday : day) : string = "Chris"
..."

This code is a lot clearer, and also more foolproof: the well-typed inputs represent exactly the
states that the function knows how to handle. Note that our requires clause is now simply true,
meaning that the function will behave properly on any well-typed input—this is a great thing to
have, and datatypes let it happen!

We could even represent the TAs as a datatype!

datatype ta = Blair | Chris | Erin | Jacob | Pankaj | Ulani
(* ohTA : day -> ta list
  * REQUIRES: true
  * ENSURES: ohTA d ==> the TA(s) holding office hours on day d
fun ohTA (Monday : day) : ta list = [Pankaj]
  | ohTA (Tuesday : day) : ta list = [Blair]
  | ohTA (Wednesday : day) : ta list = [Chris]
..."

At this point, you might be thinking that datatypes are a lot like, for example, enum types in
some other languages. But they give us so much more! Constructors can actually carry data. For
example,

datatype event = Lab of ta * ta
  | Lecture
  | OH of ta list

(* events : day -> event list
  * REQUIRES: true
  * ENSURES: events d ==> a list of events occurring in 150 on day d
  *)
...

As before, Lecture is a constructor that carries no data (i.e. Lecture : event). But, for
example, the constructor Lab takes a pair of TAs: Lab (Jacob, Pankaj) : event

You can also pattern match on constructors that carry data:
(* listTAs : event -> ta list
  * REQUIRES: true
  * ENSURES: listTAs e ==> a list of TAs involved in the event
  *)
fun listTAs (Lab (ta1, ta2) : event) : ta list = [ta1, ta2]
  | listTAs (Lecture : event) : ta list = []
  | listTAs (OH tas : event) : ta list = tas

1.1 Lists as a Datatype

Datatypes can even be recursive! You have secretly already seen a recursive datatype! Although lists are built in to ML, they could have been a datatype!

Let’s look at the inductive definition we gave for lists:

A list of integers (int list) is either

[] , or
x :: xs where x : int and xs : int list.

And that’s it!

In ML, you transcribe inductive definitions like this as datatype declarations:

datatype intlist =
  []
  | :: of int * intlist

Caveat: you can’t actually use [] as a constructor name, because the bracket notation [1,2,3] for iterating :: is built-in notation reserved for lists. But other than that, lists are just a datatype.

2 Tree Datatype

Now, we’re going to define another useful data type: integer trees

IntTrees are defined as follows:

An inttree is either

Empty, or
Node(l,x,r) where x : int and l : tree and r : tree.

And that’s it!

IntTrees can be defined as a datatype like this:

datatype inttree =
  Empty
  | Node of (tree * int * tree)

1 Maybe I’m a little too excited about this...
To see trees in action, let’s do an example:

(* REQUIRES: true
* ENSURES: size t ==> the number of elements in t
*)
fun size (t : inttree) : int =
    case t of
        Empty => 0
    | Node (l, x, r) => 1 + size l + size r
val 1 = size (Node(Empty, 1, Empty))

This illustrates structural recursion on trees: Because there are two recursive occurrences of tree as arguments to Node, you get two recursive calls, one for each.

2.1 Shrubs and recursion

Today we’re going to look at patterns of recursion on trees, but we’re going use a slightly different definition of trees. Since this datatype is not a normal tree, we’ll call it a shrub instead.

Shrubs are different from trees because there’s no data at the internal nodes of the tree, only at the leaves. There are some functions that are very natural to write for shrubs and complicated to write for trees —and vice versa—so it’s nice to know about both.

datatype intshrub =
    Leaf of int
    | Node of intshrub * intshrub

At this point in lecture, we defined two versions of a flattening operation on shrubs (one tail recursive and one not) and proved them equivalent. For details on this (and also more information about what we covered yesterday), see Mike Erdmann’s notes on structural induction.