CSC/MAT-220: Discrete Structures

Lab 8: ML

December 4, 2017

Datatype Declarations. In Lab 2 we introduced value and type bindings. We saw that these bindings allowed us documentation, but really had no affect on the behavior of the program. In contrast, the *datatype* declaration provides a means of introducing a new type that is distinct from all other types and does not merely stand for some other type. A *datatype* declaration introduces the following

i. One or more type constructors.

ii. One or more new value constructors for each of the type constructors.

The type constructors may take zero or more arguments; a zero-argument type constructor is just a type. Each value constructor may also take take zero or more arguments; a zero-argument value constructor is just a constant. The type and value constructors introduced by the declaration are new in the sense that they are distinct from all other type and value constructors. If a datatype re-defines an old type or value constructor, then the old definition is shadowed by the new one, rendering the old ones inaccessible in the scope of the new definition.

For examples, below is a datatype declaration that introduces a new type *suit* with four null value constructors: *Spades, Hearts, Diamonds, and Clubs.*

$$\textit{datatype suit} = \textit{Spades} \mid \textit{Hearts} \mid \textit{Diamonds} \mid \textit{Clubs}$$

Following this *datatype* declaration you can do a value binding, such as *val* $x =$ *Hearts*, which will create a variable x of type *suit* with value *Heart*. Note that it is conventional to capitalize the names of value constructors, but this is not required by the language.

We can also define functions on variables of type *suit*. For instance, the following function determines whether or not the first suit outranks the other in the game of bridge.

$$\begin{aligned}
&\textit{val outranks} \ : \ \textit{suit} * \textit{suit} \ \rightarrow \ \textit{bool} \ = \\
&\quad fn\ (Spades, Spades) \ => \ false \\
&\quad \mid (Spades, _) \ => \ true \\
&\quad \mid (Hearts, Spades) \ => \ false \\
&\quad \mid (Hearts, Hearts) \ => \ false \\
&\quad \mid (Hearts, _) \ => \ false \\
&\quad \mid (Diamonds, Clubs) \ => \ true \\
&\quad \mid (Diamonds, _) \ => \ false \\
&\quad \mid (Clubs, _) \ => \ false
\end{aligned}$$

Just as functions can be polymorphic, so to can datatypes. For instance, the following datatype declaration is parametrized by type $'a$.

$$\textit{datatype 'a option} = \textit{NONE} \mid \textit{SOME of 'a}$$

Here we are capitalizing our value constructors to shadow how they appear in the built in Option structure of SML. Note that *'a option* is a unary type constructor with two value constructors: *NONE* with no arguments, and some *SOME* with one argument. The values of type *typ option* are

i. The constant *NONE*, and

ii. Values of the form *SOME val*, where *val* is a value of type *typ*.

For example, some values of type *string option* are *NONE*, *SOME "Graph"*, and *SOME "Tree"*.

As noted above, the option type constructor is pre-defined in Standard ML. One common use of option types is to handle functions with an optional argument. For example, the function below computes the base b exponential for natural numbers and defaults to $b = 2$ when *NONE* is supplied as the first argument.

$$\begin{aligned}
&val\ rec\ exp\ :\ int\ option * int\ \to\ int\ =\\
&\quad fn\ (NONE, n)\ \ =>\ \ exp(SOME\ 2, n)\\
&\quad \mid (SOME\ b, 0)\ \ =>\ \ 1\\
&\quad \mid (SOME\ b, n)\ \ =>\\
&\qquad if\ n\ mod\ 2\ =\ 0\ then\\
&\qquad\quad exp(SOME\ (b*b), n\ div\ 2)\\
&\qquad else\\
&\qquad\quad b * exp(SOME\ b, n-1)
\end{aligned}$$

Another use is to handle special cases of the evaluated expression associated with a function. For example, the function below handles division by zero.

$$\begin{aligned}
&val\ divide\ :\ int * int\ \to\ int\ option\ =\\
&\quad fn\ (x, 0)\ \ =>\ \ NONE\\
&\quad \mid (x, y)\ \ =>\ \ SOME\ (x\ div\ y)
\end{aligned}$$

Other uses of option include in aggregate data structures where certain entries may be *NONE*. For instance, a cell phone number or email address in an application entry may be left blank.

Recursive Datatypes. The next level of generality is the recursive type definition. One interesting example of a recursive datatype is the natural numbers. Consider the declaration below.

$$datatype\ nat = Zero \mid Succ\ of\ nat$$

The values of type *nat* are

i. The constant *Zero*, and

ii. Values of the form Succ *nat*

The value constructor *Succ* creates the next *nat* value based on the previous value. For instance, the bindings *val* $x = Zero$ and *val* $y = Succ\ x$ creates the first and second natural number, respectively. Since *nat* is a recursive data structure, it is natural (pun intended) to defined recursive functions over variables of type *nat*. The function below doubles the value of a given natural number.

$$\begin{array}{l} val\ rec\ double\ :\ nat\ \rightarrow\ nat\ = \\ \quad fn\ Zero\ => \ Zero \\ \quad \mid Succ\ n\ => \ Succ\ (Succ\ (double\ n)) \end{array}$$

Another, far more useful, example of a recursive data structure is the binary tree.