1 Simple datatypes

We now introduce *datatype bindings*, our third kind of binding after variable bindings and function bindings.

Our first example of a datatype binding is very simple:

```ml
datatype mybool = Mytrue | Myfalse
```

This binding declares a new one-of type `mybool` and two *constructors* `Mytrue` and `Myfalse` for creating values of type `mybool`. `Mytrue` and `Myfalse` are values of type `mybool` and are the *only* values of type `mybool`. So one purpose of datatypes is to introduce new types into the language and to introduce ways of creating values of this new type.

In fact, the built-in `bool` type is defined exactly as follows:

```ml
datatype bool = false | true
```

Here's another example of a datatype binding:

```ml
datatype day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

This binding defines a new type `day` and seven new constructors for that type `Sun` – `Sat`. We can write a function that maps a day of the week to a number as follows:

```ml
fun day_to_int(d:day) =
  if d=Sun then 1
  else if d=Mon then 2
  else if d=Tue then 3
  else if d=Wed then 4
  else if d=Thu then 5
  else if d=Fri then 6
  else (*d=Sat*) 7
```

But this sequence of nested `if` expressions where we keep testing `i` is very tedious. ML provides a much more stylish way to accomplish the same thing with a `case` expression:

```ml
fun day_to_int(d: day) =
  case d of
    Sun => 1
  | Mon => 2
  | Tue => 3
  | Wed => 4
  | Thu => 5
  | Fri => 6
  | Sat => 7
```

How about mapping integers to days? Let's say that any integer outside the range `1..7` doesn't represent a valid day. So we'll use a function that returns a `day option`:
fun int_to_day(i:int) =
    case i of
      1 => SOME Sun
    | 2 => SOME Mon
    | 3 => SOME Tue
    | 4 => SOME Wed
    | 5 => SOME Thu
    | 6 => SOME Fri
    | 7 => SOME Sat
    | _ => NONE

In the final branch of that case expression, the underscore _ means “any other integer.” Notice how we can use the case expression to examine both datatypes of our own (date) as well as built-in types (int).

That might make you wonder: can we use case expressions to examine bool values? But of course. We could write, for example:

case e1 of
  true => e2
| false => e3

And that would be exactly equivalent to an if expression:

if e1
  then e2
else e3

In fact, if expressions are really just syntactic sugar for case expressions. We could have got by without ever learning if expressions. But they’re still good style to use, because they make code that tests Boolean expressions easier to understand.

So datatypes along with case expressions are powerful enough that something as important as Booleans wouldn’t even have to be a fundamental part of the language! And as we’re about to see, they’re even more powerful...

2 Compound datatypes

We start with a deliberately silly example because it will help us see the many different aspects of a compound datatype binding:

datatype mytype =
  TwoInts of int * int
| Str of string
| Pizza

This binding declares a new type whose values can contain any of the following: (1) a pair of integers or (2) a single string or (3) nothing. Any value of this type will be “tagged” with information that lets the run-time determine which variant of the type it is. The variants of mytype are TwoInts, Str, and Pizza. Those are also the names of the type’s constructors.

The binding of mytype adds four things to the static environment:

- One new type mytype that can be used just like any other type
• Three constructors: `TwoInts`, `Str`, and `Pizza`

Two of these constructors are, in fact, functions. They create values of the new type. `TwoInts` is a function of type `int*int -> mytype`, and `Str` is a function of type `string -> mytype`. `Pizza` is a value of type `mytype`.

Given a value of type `mytype`, how can we access the data stored in it? First, we have to find out which variant it is, because a value of type `mytype` might have been made from `TwoInts`, `Str`, or `Pizza` thus affecting what data is contained in the value. Once we know what variant we have, then we can access the data (if any) that the variant contains.

So how to do that?

**Idea 1: Functions.** Recall that similar problems have occurred with lists and options, which are also one-of types. (A list is either empty or an element consed onto another list. An option is either empty or some value.) For those, we have functions for testing which variant a value is (null and `isSome`) and functions for getting the data the variant contains (`hd`, `tl`, and `valOf`). Those functions raise exceptions when given arguments of the wrong variant.

ML could have taken the same approach for datatype bindings. For example, it could have taken our datatype definition above and added to the environment functions `isTwoInts`, `isStr`, and `isPizza` all of type `mytype -> bool`. And it could have added functions `getTwoInts` of type `mytype -> int*int` and `getStr` of type `mytype -> string`, which might raise exceptions. But ML does not take this approach. Instead it does something better, which we’ll discuss next.

**Idea 2: Case expressions** The better thing is a more general version of `case` expressions. Here’s a basic, silly example:

```ml
fun f (x:mytype) = 
  case x of 
    Pizza => 3 
  | TwoInts(i1,i2) => i1 + i2 
  | Str s => String.size s 
```

This example is much like our previous examples of `case` expressions, except that now we bind variables in some of the branches. In the `TwoInts` branch, variables `i1` and `i2` are bound to the integers contained in `x`. In the `Str` branch, variable `s` is bound to the string contained in `x`. Those variables can be used in the right-hand side of the branch (i.e., the expression after the arrow `=>`).

Note that we could use a `case` expression to write the functions we speculated about in the previous idea (e.g., `isTwoInts`, `getTwoInts`, etc.). But it isn’t usually idiomatic to do so. One reason for that is functional programmers prefer to write programs that won’t throw exceptions.

3 Case expressions

So far we’ve been discussing `case` expressions only by example. That’s probably how you’ve learned most features of programming languages in the past, but in this class one of our goals is to get better at learning languages. Examples are certainly useful when learning a language, but we can get a much clearer idea of how language features work when we describe their syntax and semantics. Let’s do that, now, for `case` expressions.

The **syntax** of a `case` expression is as follows:

```ml
case e of b1 | b2 | ... | bn 
```
Each of the \( b_i \) is a \textit{branch} of the \textit{case}. Each branch is separated with the vertical bar character \( \mid \), pronounced “or”.

The syntax of a branch \( b \) is \( p \Rightarrow e \) where \( p \) is a \textit{pattern} and \( e \) is an expression. We won’t give the full syntax for patterns yet. For now, let’s just say that a pattern “looks like” a constructor call, or is an underscore \( _{} \). Patterns are \textbf{not} expressions: we can’t use all the rich expression syntax we’ve developed inside a pattern.

The \textbf{type checking rule} for \textit{case} expressions requires two things, for now. First, \( e \) and all the patterns \( p_1 \ldots p_n \) must have the same type \( t_a \). Second, all the branch expressions \( e_1 \ldots e_n \) must have the same type \( t_b \). Note how this rule generalizes the rule for \textit{if} expression, which required \( t_a \) always to be \texttt{bool}.

The \textbf{evaluation rule} for \textit{case} expressions first evaluates \( e \) to a value \( v \). Then \( v \) is \textit{matched} against each pattern \( p_1 \ldots p_n \) in order. Therefore, evaluation of a \textit{case} expression is called \textit{pattern matching}. Suppose that the first matching pattern is \( p_j \). Then expression \( e_j \) is evaluated to a value \( v_j \). That value becomes the result of evaluating the entire \textit{case} expression. If no matching pattern is found, an exception is raised. So at most one branch expression is ever evaluated.

For this lecture, we’ll postpone an exact description of the full syntax, type checking, and evaluation rules for pattern-matching and just keep things simple. For now, let’s say that each branch pattern uses a different constructor, and pattern-matching evaluation chooses the branch that “looks like” value \( v \). The \textit{wildcard} pattern, written as underscore \( _{} \), “looks like” any value. In our running example, if value \texttt{TwoInts(7,9)} is passed to \( f \), then the second branch will be chosen, because its pattern “looks like” that value. That takes care of the “check the variant” concern of using one-of types.

Pattern matching also takes care of the “access the contained data” concern of one-of types. For example, \texttt{TwoInts} contains two integer values. Therefore a pattern for it can (and, for now, must) bind two variables, which are \((i_1,i_2)\) in our running example. As part of matching, the corresponding parts of the value (7 and 9) are bound to \( i_1 \) and \( i_2 \) in the dynamic environment used to evaluate the corresponding branch expression (the \( i_1+i_2 \)). In this sense, pattern matching is like a \texttt{let} expression. It binds variables in a local scope. The type checker knows what types these variables must have, because those types were specified in the datatype binding that created the constructor used in the pattern.

\textbf{Idea 1 vs. Idea 2.} Why are \textit{case} expressions better than functions for testing variants and extracting data? Here’s one of the best reasons: If a programmer accidentally omits a variant from a \textit{case} expression, the compiler will issue a warning message. This message indicates that evaluating the \textit{case} expression could fail to find a matching branch, in which case it will raise an exception. If the compiler does not generate such a warning, the programmer is guaranteed that no exception will be raised. This is \textit{wonderful}: the compiler is guaranteeing that a program will never exhibit a particular kind of bug. (This is one of the main reasons we, in grading, treat warning messages as errors. We want you to learn to take full advantage of this language feature.)

Here are some other reasons:

- We can never “mess up” and try to extract data from the wrong variant. For example, we can’t try to extract two integers from the string variant in \texttt{mytype}.
- If a \textit{case} expression uses a variant twice, then the type-checker will give an error message, because one of the branches could never possibly be used—which means the programmer probably didn’t write the code they meant to write. Again, the compiler is helping to prevent bugs.
- If you still want functions like \texttt{null} and \texttt{hd}, you can easily write them yourself by using pattern matching. (But except for some specific purposes that we’ll discuss later, this is not idiomatic in functional programming.)

Pattern matching is so massively useful, general, and powerful that it will be the subject of the entire next lecture.
4 Useful examples

Let us now consider several examples where “one-of” types are useful, since so far we considered only a silly example.

As we saw at the beginning of the lecture (with days), they are good for enumerating a fixed set of options—and are much more idiomatic than using, for example, integers. Here’s another example:

datatype suit = Club | Diamond | Heart | Spade

Many languages have support for this sort of enumeration including Java and C, but ML takes the next step of letting variants contain data:

datatype rank = Num of int | Jack | Queen | King | Ace

We can then combine the two pieces with an each-of type to represent a card:

val jack_of_clubs = (Jack, Club)
val royal_club_flush = [(Num 10, Club), jack_of_clubs, (Queen, Club), (King, Club), (Ace, Club)]

One-of types are also useful when different situations call for different data. For example, suppose you want to identify students by their id numbers, but when students do not yet have an id number (perhaps because they are new to the university), then you will use their full name instead. This datatype binding captures the idea directly:

datatype student_id =
  IdNum of int
| FullName of string

Don’t Use Each-of When You Mean One-of. Unfortunately, programmers who haven’t studied principles of programming languages often show a profound lack of understanding of one-of types vs. each-of types. They end up using each-of instead of one-of. That’s like using a screwdriver to do the job of hammer. It can work, but you’re using the wrong tool. Consider this record type:

(* If id_num is ~1, then use the name field,
  * otherwise ignore the name field *)
{id_num : int, full_name : string}

Any code that uses this type has to check for ~1. The programmer has to remember that—she gets no help from the compiler. So there are likely to be bugs. Furthermore, this type wastes space, because every record has one field that is unused.

On the other hand, each-of types along with option types are exactly the right approach if we want to store for each student both their id number, if they have one, and their full name:

{ id_num : int option, full_name : string }

5 A datatype for trees

Here’s a datatype for binary trees that contain integers at each leaf and node:
datatype tree =
  Leaf of int |
  Node of tree * int * tree

This datatype exemplifies all the kinds of compound types: it is one-of (because ever element of the tree is either a Leaf or a Node), it is each-of (because every Node contains a tree, an int, and another tree), and is it self-reference (because trees contain trees).

We can write a very elegant function to determine whether a tree contains a particular integer:

fun contains(t:tree, i:int) =
  case t of
    Leaf(j) => i=j |
    Node(l,j,r) => i=j orelse contains(l,i) orelse contains(r,i)

6 A datatype for expressions

Our last example is a datatype for arithmetic expressions containing constants, negations, additions, and multiplications.

datatype exp = Constant of int |
  Negate of exp |
  Add of exp * exp |
  Multiply of exp * exp

This datatype is a self-reference type. It describes trees in which the leaves are integers and the internal nodes are either negations with one child, additions with two children, or multiplications with two children. Here’s a function that evaluates an exp in the same way that ML would evaluate an ML expression:

fun eval (e:exp) =
  case e of
    Constant i => i |
    Negate e1 => ~ (eval e1) |
    Add(e1,e2) => (eval e1) + (eval e2) |
    Multiply(e1,e2) => (eval e1) * (eval e2)

This function call evaluates to 15:

eval (Add (Constant 19, Negate (Constant 4)))

Notice how we can nest constructors. Constructors are just functions, so this isn’t a new feature; it was implied by the previous rules we had for evaluating function calls.

There are many functions we might write over values of type exp. Most of them will use pattern-matching and recursion in a similar way. For examples, we could compute:

- The largest constant in an expression.
- A list of all the constants in an expression (using list append).
- A bool indicating whether there is at least one multiplication in the expression.
The number of addition expressions in an expression.

Here’s how to write that last function:

```haskell
fun number_of_adds e =
  case e of
    Constant i => 0
  | Negate e1 => number_of_adds e1
  | Add(e1,e2) => 1 + number_of_adds e1 + number_of_adds e2
  | Multiply(e1,e2) => number_of_adds e1 + number_of_adds e2
```

Notice how we lined up the arrows => in this code. It’s nice style to do so in short functions, though not strictly necessary.