Datatypes

COMP 302

PART 06

Datatype Bindings

We have seen

- Variable bindings
- Function bindings

We now introduce datatype bindings in which we associate the name of a datatype with the possible values

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

This declares a new type (mybool) and two constructors for creating values of this type, Mytrue and Myfalse which are the only values of this bype

(Note that the type bool is actually defined as datatype bool = true | false)

(This is similar to enum types in Java, C, etc)

Simple Datatype Bindings

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

(By convention, we begin the names of type constructors with uppercase letters)

Defining a function on this datatype

```
fun day_to_int(d : day) : int =
  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
```

Simple Datatype Bindings

For user defined datatypes we generally use case expressions in accessing values

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

Simple Datatype Bindings

```
fun int_to_day(i: int) : day =
   case (i mod 7) + 1  of
     1 => Sun
   | 2 => Mon
   | 3 => Tue
   | 4 => Wed
   | 5 => Thu
   | 6 => Fri
   | _ => Sat
```

Type Synonyms

Another kind of binding is a type synonym

For example

```
type intPairList = (int * int) list
```

This binding just gives a name for the type and is interchangeable with the type.

The name is added to the static environment for future type-checking

Playing Cards

```
datatype suit = Hearts | Diamonds | Spades | Clubs

datatype rank = Ace | King | Queen | Jack | Ten | Nine | Eight | Seven | Six |
Five | Four | Three |Two

type card = rank * suit

val c1 = (Queen, Hearts)
```

Functions on datatypes

In games such as bridge, certain suits outrank others:

```
fun outranks (s1 : suit, s2 : suit) : bool =
  case (s1, s2) of
        (Spades, Spades) => false
     |  (Spades,_) => true
     |  (Hearts, Spades) => false
     |  (Hearts, Hearts) => false
     |  (Hearts,_) => true
     |  (Diamonds, Clubs) => true
     |  (Diamonds,_) => false
     |  (Clubs,_ ) => false
```

Constructors in Datatypes

Instead of just having constants in datatype definitions, we can define constructors which take an argument.

They have the form: `Name of type`

The constructor with the given name can take arguments of the given type to form values of the datatype

Geometric Figures

```
datatype color = Red | Green | Blue

datatype shape =
   Circle of color * real
  | Rectangle of color * real * real
```

This adds new types color and shape to the environment

It adds constructors Circle and Rectangle

A constructor is (among other things) a function

It can create values of the new type

Data Constructors

Data constructors can be used to create values of the new type.

```
val c = Circle (Red, 2.0)
```

a value of type shape is made from one of the constructors

the value contains

- a tag for which constructor was used (Circle)
- the actual data (Red, 2.0)

Structural Recursion

Data constructors in functions to discriminate between variants and decompose them. This allows structural recursion in function definitions.

```
fun area s =
  case s of
      Circle (_, r) => Math.pi * r * r
    | Rectangle (_, len, wid) => len * wid

val a = area c
```

Note: type of area is shape -> int

Methodology

To access a value of a certain datatype

- check which variant it is (which constructor created it)
- extract the data if there is any

Case

- in the area function we have multi-branch conditional to find branch based on variant
- we extract the data and bind it to variables local to that branch

Pattern Matching

Syntax of the case statement where pi is a pattern:

```
case e0 of
      p1 => e1
    | p2 => e2
       . . .
    | pn => en
```

- each pattern is a constructor followed by the right number of variable “arguments”
- most patterns look like expressions but aren’t
- we don’t evaluate patterns
- we just see whether the value of e0 matches them

Finding the Maximum

Finding the maximum value in a list of integers:

```
fun max (lst : int list) : int =
  case lst of
    [] => 0     (* Is that right? *)
  | x :: [] =>  x
  | x :: xs => let val y = max(xs)
               in
                  if x > y then x else y
               end
```

How about using options?

The example above returns 0 if the list is empty

This is arbitrary and incorrect

We could raise an exception (to be covered later)

Can use options:

An option contains 0 or 1 objects

An option with 0 objects is NONE

An option with 1 object is accessed by SOME

Using Options to avoid arbitrary 0

```
fun max (lst : int list) int option =
  case lst of
     [] => NONE
   | x :: xs =>  let y = max (xs)
                 in
                   case y of
                      NONE => x
                      SOME v => if x > v then SOME x else y
                 end
```

The Option Datatype

Options are build into SML but can be defined using our standard datatype constructors:

```
datatype intoption =
    NONE
  | SOME of int

val x : intoption = SOME 17
```

Recursively Defined Datatypes

Lists are predefined in SML but we can also define our own lists:

```
datatype intlist = Nil | Cons of (int * intlist)
```

There are two constructors, Nil and Cons

It is recursive because it uses its own definition in the Cons constructor

Examples:

```
val list1 = Nil
val list2 = Cons(2, Cons(1, Nil))
val list3 = Cons(5, list2)
```

Functions on Lists

We can use pattern matching to decompose user defined recursive datatypes.

Note that every function uses a case with the same pattern

```
fun length (lst : intlist) : int =
  case lst of
    Nil => 0
  | Cons (h, t) => 1 + length (t)

fun isEmpty (lst : intlist) : bool =
  case lst of
    Nil => true
  | Cons (_, _) => false
```

Functions on Lists

```
fun append (lst1 : intlist, lst2 : intlist) : intlist =
  case lst1 of
    Nil => lst2
  | Cons (h, t) => Cons (h, append (t, lst2))

fun reverse (lst : intlist) : intlist =
  case lst of
    Nil => Nil
  | Cons (h, t) => append (reverse(t), Cons(h, Nil))
```

Binary Trees

Unlike lists, binary trees are not build into SML.

However, they can be defined using data types:

```
datatype tree =
   Empty
| Node of tree * int * tree
```

```
var smalltree = Node (Node (Empty, 1, Empty), 3, Node(Empty, 2, Empty))
```

Functions on Trees

Functions that operate on trees can be defined using pattern matching:

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

Short circuit logical operators

Boolean Operators

Syntax: e1 andalso e2

Type checking:

- e1 and e2 must both have type bool

Evaluation:

- if e1 evaluates to false then false else result of evaluating e2

Note: This is short circuit evaluation (unlike many other languages).

That implies that andalso and orelse are not functions since they do not use call-by-value semantics

Short circuit logical operators

We also have:

e1 orelse e2

not e

not is a pre-defined function

The common operators &&, || don’t exist in ML. ! exists but does not mean “not”

These operators are syntactic sugar. They can be replaced by if expressions:

- if e1 then e2 else false equivalent to e1 andthen e2
- if e1 then true else e2 equivalent to e1 orelse e2
- if e1 then false else true equivalent to not e1

Expressions

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

Evaluating an 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)

val fifteen = eval (Add (Constant 19, Negate (Constant 4)))
```

Other Functions on Expressions

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