Core SML

CS 131: Programming Languages
September 6, 2001

Review

- Last class you saw lots of types:
  - Base types:
    ```
    int        real       bool
    char       unit       string
    ```
  - Product types:
    ```
    int*bool   real*int*string
    int*int    etc.
    ```
  - Function types
    ```
    int->int                    (real*int*int)->bool
    (int*bool)->(bool*int)      etc.
    ```
  - List types
    ```
    int list                    (int*bool) list
    (int list) list             etc.
    ```

Review

- You saw ways to bind variables to values:

```
val x    = [3+4, 5+6]
val succ = (fn x => x+1)
fun succ(x) = x+1
val succ' = succ
```

- You saw pattern-matching and "clausal definitions"

```
fun prod [] = 1
  | prod (n::ns) = n * (prod ns)
```

Multi-Argument Functions

- Every function takes exactly one argument, but that argument might be a tuple (or a record)

```
fun power(x,n) =
   if (n = 0) then
        1.0
   else
        x * power(x,n-1)
```

- What's the type of `power` ?
- Alternate definition using pattern matching?

Let-Expressions

- Method of local variable declarations
- Have the form
  `let <definitions> in <expression> end`
- Evaluation process:
  - Evaluate definitions in sequence, binding any variables
  - Evaluate the expression (the "body" of the **`let`**)
  - Forget the new variable bindings
  - Return the value of the body

Example

```
fun solve_quadratic(a,b,c) =
  let
     val disc = b*b - 4.0*a*c
     val sqrtdisc = Math.sqrt disc
     val denom = 2.0*a
  in
     ((~b + sqrtdisc) / denom,
      (~b - sqrtdisc) / denom)
  end
```

What is the type of this function?

Length of a List

- Length function for integer lists:

```
fun length []       = 0
  | length (_::xs) = 1 + length xs
```

- Better definition:

- What is the type of `length` ?

Length of a List

- We said SML could infer types omitted by the programmer

```
fun length []       = 0
  | length (_::xs) = 1 + length xs
```

- What is the type of `length` now ?

Types of the Empty List

- Note that
```
[]  :  int list
[]  :  bool list
[]  :  ((string * string) -> string) list
[]  :  (string * (string -> string)) list
```
- In fact, for any type *t* we have:
```
[]  :  t list
```

Types of `length`

```
fun length []      = 0
  | length (_::xs) = 1 + length xs
```

- So, for any type *t*, it is reasonable to say

```
length  :  t list -> int
```

Polymorphic Types

- SML permits variables representing types, written with a leading prime
  - For example, `'a` or `'b` or `'c`
- Then we can say
```
[]     : 'a list
length : 'a list -> int
```
- Type variables in such types are implicitly universally-quantified

More Polymorphic Functions

```
fun identity x = x

fun diagonal x = (x,x)

fun swap(x,y) = (y,x)

fun append([],ys) = ys
  | append(x::xs, ys) = x :: append(xs,ys)
```

Datatypes

- The **`datatype`** mechanism generalizes:
  - Enumerated types
  - Tagged unions
  - Inductive types (lists, trees, etc.)
- Provides facility for constructing data structures and doing pattern matching

Enumerated Types

```
datatype day =
  Sunday | Monday | Tuesday | Wednesday |
  Thursday | Friday | Saturday
val weekdays : day list =
     [Monday, Tuesday, Wednesday,
      Thursday, Friday]
fun isWeekend Saturday = true
  | isWeekend Sunday = true
  | isWeekend _ = false
```

Tagged Unions

- Suppose we want a list that can contain both integers and reals. First, define:

```
datatype num = I of int
             | R of real
```

- Then

```
I 5    :  num
R 5.0  :  num
R 3.1  :  num
```

Tagged Unions

```
val mylist : number list =
  [I 3, R 4.0, R 1.1, I(5+5), I ~17]
```

Note: The addition operator + does not work on `num`'s!

Tagged Unions

```
fun addnums (I n, I m) =  I(n+m)
  | addnums (R r, R s) =  R(r+s)
  | addnums (I n, R s) =  R((Real.fromInt n) + s)
  | addnums (R r, I m) =  R(r + (Real.fromInt m))
```

Tags and Enumerations

```
datatype color = Red | Orange | Yellow | Green
               | Blue | Indigo | Violet
               | RGB of int*int*int
```

```
Red             : color
Indigo          : color
RGB(25,25,25) : color
```

Inductive Types

```
datatype itree = IEmpty
               | INode of itree*int*itree

val t1 : itree = INode(IEmpty, 3, IEmpty)
val t2 : itree = INode(t1, 7, IEmpty)
val t3 : itree = INode(t1, 8, INode(t1, 9, t2))
```

Using Trees

```
fun sumtree IEmpty = 0
  | sumtree (INode(left,n,right)) =
      (sumtree left) + n + (sumtree right)
```

```
fun member (n, IEmpty) = false
  | member (n, INode(left,m,right)) =
      if (n = m) then
         true
      else if (n < m) then
         member(n, left)
      else
         member(n, right)
```

Using Trees

Exercise: Write the function

```
insert : int * itree -> itree
```

for ordered trees.

Arithmetic Expressions

```
datatype exp = Num of real
             | Sum of exp*exp
             | Diff of exp*exp
```

```
Num 4.0                                 :  exp
Sum(Num 3.0, Diff(Num 4.0, Num 1.0))    :  exp
```

Exercise: define the function `eval : exp -> real`

Type Abbreviations

- The datatype construct creates a *new* type
- We can also give shorter names to existing types

```
type ip = int * int
type bp = bool * bool
```

- Then `ip` is synonymous with `int*int`
- Similarly `bp` is interchangeable with `bool*bool`

Type Abbreviations with Parameters

- Definitions of types can be *parameterized*

```
type 'a pair = 'a * 'a

val p1 : int pair = (3,4)
val p2 : ip = p1
val p3 : bool pair = (true,true)
```

Datatypes with Parameters

- Datatypes can also be parameterized

```
datatype 'a tree =
        Leaf of 'a
        Node of ('a tree) * ('a tree)

val t1 : int tree = Leaf 5
val t2 : bool tree =
      Node(Leaf true,Leaf false)
```

Datatypes with Parameters

```
datatype 'a tree =
   Leaf of 'a
   Node of ('a tree) * ('a tree)

fun collect (Leaf x) = [x]
  | collect (Node(left,right)) =
       (collect left) @ (collect right)
```

Predefined Datatypes: `option`

```
datatype 'a option = NONE
                   | SOME of 'a
```

```
NONE           : int option
SOME 3         : int option
SOME 12        : int option

Int.fromString : string -> int option
```

Predefined Datatypes: `list`

```
datatype 'a list = nil
                 | :: of ('a * 'a list)

infix ::
```

```
nil          : int list
3::(4::nil) : int list
```

(The `[x,y,z]` notation is built-in magic, however.)