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.

• 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
  \[
  \text{let} \ <\text{definitions}> \ \text{in} \ <\text{expression}> \ \text{end}
  \]
• Evaluation process:
  - Evaluate definitions in sequence, binding any variables
  - Evaluate the expression (the "body" of the \text{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 sqrdisc = Math.sqrt disc
    val denom = 2.0*a
  in
    ((-b + sqrdisc) / denom, (-b - sqrdisc) / denom)
  end

What is the type of this function?

Length of a List

• Length function for integer lists:

\[
\text{fun} \ length \ [] = 0 \\
| \ length \ (\_::xs) = 1 + \text{length} \ xs
\]

• Better definition:

• What is the type of \text{length} now?

Length of a List

• We said SML could infer types omitted by the programmer

\[
\text{fun} \ length \ [] = 0 \\
| \ length \ (\_::xs) = 1 + \text{length} \ xs
\]

• What is the type of \text{length} now?
Types of the Empty List

• Note that
  
  \[
  [] : \text{int list} \\
  [] : \text{bool list} \\
  [] : ((\text{string} \times \text{string}) \rightarrow \text{string}) \text{ list} \\
  [] : (\text{string} \times (\text{string} \rightarrow \text{string})) \text{ list} \\
  \]

• In fact, for any type \( t \) we have:
  
  \[
  [] : t \text{ list} \\
  \]

Types of \text{length}

\[
\text{fun} \quad \text{length} [ ] = 0 \quad \| \quad \text{length} (_::xs) = 1 + \text{length} \ xs \\
\]

• So, for any type \( t \), it is reasonable to say

\[
\text{length} : t \text{ list} \rightarrow \text{int} \\
\]

Polymorphic Types

• SML permits variables representing types, written with a leading prime
  
  - For example, ‘a or ‘b or ‘c

• Then we can say
  
  \[
  [] : \text{‘a list} \\
  \text{length} : \text{‘a list} \rightarrow \text{int} \\
  \]

• Type variables in such types are implicitly universally-quantified

More Polymorphic Functions

\[
\text{fun} \quad \text{identity} \ x = x \\
\text{fun} \quad \text{diagonal} \ x = (x,x) \\
\text{fun} \quad \text{swap}(x,y) = (y,x) \\
\text{fun} \quad \text{append}([],ys) = ys \\
\quad \text{append}(x::xs, ys) = x :: \text{append}(xs,ys) \\
\]

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 day =
  | 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

```haskell
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

```haskell
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

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

Using Trees

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

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

\[
\text{insert : int * itree -> itree}
\]

for ordered trees.

Arithmetic Expressions

\[
\text{datatype exp = Num of real} \\
\quad | \text{Sum of exp*exp} \\
\quad | \text{Diff of exp*exp}
\]

\[
\begin{align*}
\text{Num 4.0} & : \text{exp} \\
\text{Sum(Num 3.0, Diff(Num 4.0, Num 1.0))} & : \text{exp}
\end{align*}
\]

Exercise: define the function \(\text{eval : exp -> real}\)

Type Abbreviations

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

\[
\text{type ip = int * int} \\
\text{type bp = bool * bool}
\]

- Then \(\text{ip}\) is synonymous with \(\text{int*int}\)
- Similarly \(\text{bp}\) is interchangeable with \(\text{bool*bool}\)

Type Abbreviations with Parameters

- Definitions of types can be parameterized

\[
\begin{align*}
\text{type } \alpha \text{ pair } &= \alpha \times \alpha \\
\text{val p1 : int pair} &= (3,4) \\
\text{val p2 : ip} &= \text{p1} \\
\text{val p3 : bool pair} &= (\text{true,true})
\end{align*}
\]
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)
```

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.)