Computer Science 131
Programming Languages

August 29, 2000
Advanced Core SML
Review

• In the last lecture, you saw...

• Lots of types
  - Base types: int, bool, real, string, char, unit
  - Product types, e.g., int*bool
  - Function types, e.g., int->int and real*int->int
  - List types, e.g., int list and (int*int) list
• Ways to bind variables to values
  
  val x = 3 + 4
  fun succ(x) = x+1

• Pattern-matching and clausal definitions
  
  fun power(x, 0) = 1.0
  
  | power(x, n) = x * power(x, n-1)

  fun prod [] = 1
  
  | prod (n::ns) = n * (prod ns)
Length of a list

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

- What is the type of length?
Types of the Empty List

• Note that
  
  \[
  [] : \text{int list}
  \]
  
  \[
  [] : \text{bool list}
  \]
  
  \[
  [] : (\text{string}*\text{string} \rightarrow \text{string}) \text{ list}
  \]
  
• In fact, for any type \( t \), we have
  
  \[
  [] : t \text{ list}
  \]
Types of length

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

• Similarly, for any type t, we have
  length : t list -> int

• Note: here t is used as a variable ranging over types.
Polymorphic Types

• SML has variables representing types
  – 'a and 'b and 'c etc.

• Then we can say
  
  [ ] : 'a list
  length : 'a list -> int

• Type variables are implicitly universally-quantified
Polymorphic Examples

- What are the types of these functions?
  
  ```haskell
  fun identity x = x
  fun diag x = (x,x)

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

  fun append([],ys) = ys
  | append(x::xs, ys) = x :: append(xs,ys)
  ```
Functions and Re-binding

• Consider the following code:
  
  ```
  val x = 3
  fun add_x (y:int) = y+x
  val x = 7
  ```

• Now, what is the value of `add_x(2)`?
Functions and Re-binding

• Consider the following code:
  val x = 3
  fun add_x (y:int) = y+x
  val x = "some string"

• Now, what is the value of \( \text{add}_x(2) \) ?
Static Scope

val x = 3
fun add_x (y:int) = y+x
val x = "some string"

• Here \( x \) is a free variable of the function
• SML uses static scoping
  - Summary: Bindings of free variables are stored when the function is created.
Functions that return functions

• Consider the following functions:

  fun add1 (x) = x + 1
  fun add2 (x) = x + 2
  fun add7 (x) = x + 7

  Can we generalize this?

• Define a function that, given \( n \), returns the function which adds \( n \) to its argument
Functions that return functions

• Define a function that, given $n$, returns the function which adds $n$ to its argument

    fun make_adder (n:int) = fn x => n+x

    make_adder : int -> (int -> int)
    make_adder : int -> int -> int
Functions that return functions

fun make_adder (n:int) = 
  fn x => n+x

make_adder : int -> (int -> int)

• Then we can say
  val succ = make_adder 1
  val add7 = make_adder 7
Functions that return functions

• SML has special syntax for such functions.
  
  ```sml
  fun make_adder (n:int) = 
    fn x => n+x
  ```

  can be written as

  ```sml
  fun make_adder n x = n+x
  ```

• Note space between arguments, and the =
Function Composition

• Write a function \texttt{compose} that, given functions \( f \) and \( g \) returns their composite.

• Recall: composite of \( f \) and \( g \) is the function which maps \( x \) to \( f(g(x)) \).
Function Composition

• Write a function \texttt{compose} that, given functions \( f \) and \( g \) returns their composition

\[
\text{fun compose (f,g) } = \\
\hspace{1cm} \text{fn x => f(g(x))}
\]

\[
\text{fun compose (f,g) x = f(g(x))}
\]

• What is the type of \texttt{compose}?
Applying a Function to a List

• Problem: define a function map that
  - takes as its argument a function $f$
  - returns a function that given a list applies $f$ to every element of that list.
  - that is,
    
    $(\text{map } f) \ [x_1, \ldots, x_n] = [f(x_1), \ldots, f(x_n)]$
Applying a Function to a List

• Suppose you have the function $f$.
  - How do you define a new function that applies $f$ to every element of a list?
Applying a Function to a List

• Suppose you have a function $f$.
  - How do you define a new function that applies $f$ to every element of a list?

fun loop [] = []
  | loop (x::xs) = (f x) :: (loop xs)
Applying a Function to a List

• Then \texttt{map} is just the function which takes \( f \) and returns that looping function.

\[
\text{fun map f =}
\text{let}
\quad \text{fun loop [] = []}
\quad \quad \mid \text{loop (x::xs) = (f x) :: (loop xs)}
\text{in}
\quad \text{loop}
\text{end}
\]
Applying a Function to a List

• A different way of writing the same function:

```plaintext
fun map f [] = []
    | map f (x::xs) = (f x) :: ((map f) xs)
```

• Compare:

```plaintext
fun map (f,[]) = []
    | map (f,x::xs) = (f x) :: (map (f,xs))
```
Datatypes

• The *datatype* mechanism generalizes:
  - enumerated types
  - (tagged) unions
  - inductive types
    • e.g., lists, trees, etc.
Enumerated Types

datatype day =
    Sunday | Monday | Tuesday | Wednesday |
    Thursday | Friday | Saturday

val weekdays : day list =
    [Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday]

fun isWeekend Saturday = true
    | isWeekend Sunday = true
    | isWeekend _ = false
Tagged Unions

• Suppose we want numbers that can be integers or reals
  
  ```markdown
datatype num =
    INT of int | REAL of real
  ```

• Then
  
  ```markdown
INT 5 : num
REAL 5.0 : num
  ```

  ```markdown
INT : int -> num
REAL : real -> num
  ```
Tagged Unions

datatype num =
    INT of int | REAL of real

fun addnum (INT n, INT m) = INT(n+m)
| addnum (REAL r, REAL s) = REAL(r+s)
| addnum (INT n, REAL r) =
    Real((Real.fromInt n) + r)
| addnum (REAL r, INT n) =
    Real(r + (Real.fromInt n))
Trees

datatype ttree = TLeaf
           | TNode of ttree*ttree

TLef : ttree
TNode(TLeaf,TLeaf) : ttree
TNode(TLeaf,TNode(TLeaf,TLeaf)) : ttree
Trees with Data

datatype itree = ILeaf of int
            | INode of itree*itree

ILeaf 3 : itree

INode(ILeaf 4, ILeaf 5) : itree
Trees with Data

datatype itree = ILeaf of int
  | INode of tree*tree

fun sumtree (ILeaf n) = n
  | sumtree (INode(left,right)) =
      (sumtree left)+(sumtree right)
Arithmetic Expressions

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

fun eval (Num r) = r
  | eval (Sum(e1,e2)) =
    (eval e1) + (eval e2)
  | eval (Diff(e1,e2)) =
    (eval e1) - (eval e2)
Type Definitions

• Define abbreviations for types using `type`

  type intpr = int * int
  type boolpr = bool * bool

• Then `intpr` is synonymous with `int*int`

• Similarly `boolpr` is interchangeable with `bool*bool`
Type Definitions with Parameters

- Definitions of types can be parameterized
  
  ```
  type 'a pair = 'a * 'a
  ```

- Then
  
  - string pair is synonymous with
    ```
    string*string
    ```
  
  - int pair = int*int = intpr
Datatypes with Parameters

• Definitions of types can be parameterized

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

• Then

```haskell
Leaf 5                     :  int tree
Node(Leaf true,Leaf false) :  bool tree
```
Datatypes with Parameters

• Definitions of types can be parameterized

```ml
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)
```
Defining list

- The list type is definable in SML

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

where `::` is then made infix

- The nice `[x, y, z]` notation is magic though.