SML-2

- User defined types
- Functions on user defined types
- Higher order functions
  – Reduce
  – Fold

Strictness versus Eagerness

- SML is an eager PL. All arguments to a function are evaluated on entry.
- A function is strict in an argument if it always uses that argument; then it's safe to evaluate that argument on entry.
- Some PLs use lazy evaluation, delaying argument evaluation until it's necessary for the value to be used.
- Compile-time analysis for strictness allows optimization

Reduce

- **Powerful higher level function that works on lists in a process termed *list reduction***
- **If f is to be defined on the elements of the list, u the result for the empty list and $ a binary operation then:**

  f nil = u

  f [x1, x2, x3,…,xk] = x1 $ x2 $ x3 $ … $ xk

  Associativity of $ counts here as to how this is evaluated.

  Need identity element for $ operation, such as :

  plus, + (0), times, * (1), append, @ (nil)

Typing Reduce

```
>fun reduce f  zero  [ ] = zero |  (* 1 *)
  reduce f  zero (x::tl) = f  x  (reduce f  zero tl); (* 2 *)
val reduce = fn :(‘a → ‘b → ‘b) → ‘b → ‘a list → ‘b
```

- **Let's see how to type reduce.**
  - **Most general type is** *fn :(‘a → ‘b → ‘c) → ‘d → ‘e list → ‘f*
  - **From (2) can see x is 1st argument of f, so ‘e = ‘a;**
  - **From (2) also, result of reduce is 2nd argument of f, so ‘f = ‘b yielding simpler type for reduce:**

    fn :(‘a → ‘b → ‘c) → ‘d → ‘a list → ‘b
  - **From (1), zero is return type of reduce so ‘d = ‘b and f is return type of reduce function from (2) so ‘c = ‘b, yielding final type for reduce:**

    fn :(‘a → ‘b → ‘b) → ‘b → ‘a list → ‘b

Reduce

Recall, add x y:int = x+y.

>reduce add 0 [1,2,3,~1,~2,~3];

val it = 0 : int

>fun times x y: int = x * y;

val times = fn: int → int → int

>reduce times 1 [3,1,2,~1,~2,~3];

val it : ~36: int

```
fun reduce f  zero  [ ] = zero |
 reduce f  zero (x::tl) = f  x  (reduce f  zero tl);
```


Reduce

- **Intuition: think of the list as constructed from elements at the top level by cons operations (that is, :: constructions)**

  cons 7 (cons 3 (cons 13 nil)) = [7,3,13]

  Then the effect of *reduce add 0 [7,3,13];*

  is to insert *add* for the "cons-es" in its list operand and *0* for *nil* in that list, that is:

  add 7 (add 3 (add 13 0))

  Note: this is doing addition using right associativity, so you need to be careful about the operator you use with reduce, as it has to be right associative.

Using *reduce* to build *flatten*

- Can build a list *flatten* using list append function

  >fun app nil y = y | app (x::xs) y = x::(app xs y);

  val app = fn: ‘a list → ‘a list → ‘a list

  >fun flatten xs = reduce app nil xs;

  val flatten = fn: ‘a list list → ‘a list

  >flatten ([[1,2], [3,4]]);

  val it = [1,2,3,4]: int list

  >*flatten ([[[1]], [[2]]]);*

  val it = [[1],[2]]: int list list

  What’s wrong here? Flatten is only working on the top level elements of the list. How would you build a list flatten that flattens every sublist?

flatten Example

Think of replacing all top level “cons-es” by the function argument, app

flatten ([[1,2], [3,4]]); Ans: [1,2,3,4]

reduce app nil (cons [1,2] (cons [3,4] [ ]))

app app

flatten [[[1]],[[2]]]; Ans: [[1],[2]]

reduce app nil (cons (cons (cons 1 nil), nil),

app [1] ⟹ [[1]]

(cons (cons 2, nil), nil))

app [2] ⟹ [[2]]

app [[1]] [[2]] ⟹ [[1],[2]]

Building *map* from *reduce*

>fun cons x y = x::y;

val it = fn: ‘a → ‘a list → ‘a list;

>fun comp f g x = f (g (x));

val comp = fn: (‘a → ‘b) → (‘c → ‘a) → ‘c → ‘b

>fun mymap f llist = reduce (comp cons f) [ ] llist;

val it = fn: (‘a → ‘b) → ‘a list → ‘b list

> fun incr = add 1;

val it = fn: int → int

> mymap incr [12,2];

val it = [13,3]: int list

mymap Example

mymap incr [12,2] = mymap incr (cons 12 (cons 2 nil))

=((comp cons incr) 12 ((comp cons incr) 2 nil))

=((cons (incr 12)) (cons (incr 2) nil))

(cons 3 nil))

= (cons 13 (cons 3 nil))

= [13, 3].

Fold1

A left associative reduction operation

```
>fun fold1  f  u  nil = u |  fold1  f  u  (x::xs) =
    fold1  f ( f  u  x)  xs;
```

val fold1 = fn: (‘a → ‘b → ‘a) → ‘a → ‘b list → ‘a

fold1 uses its 2nd parameter as an accumulator for the partially calculated value, initializing it to the identity for the f operation

```
>fold1  append  nil  [[1], [2], [3]]; (*another list flatten*)
```

val it = [1,2,3] :int list

```
>fold1 add 1 [ 1,2,3];
```

val it = 7 : int

fold1 versus *reduce*

```
=fold1 add 0 [7,3,13]  (*better storage behavior*)
=fold1 add 7 [3,13]
=fold1 add 10 [13]
=fold1 add 23 nil
=23
=reduce add 0 [7,3,13]
=add 7 (reduce add 0 [3,13])
=add 7 (add 3 (reduce add 0 [13])
=add 7 (add 3 (add 13 0)) = add 7 (add 3 (13))
=add 7 16 = 23
```

```
fun fold1  f  u  nil = u |
   fold1  f  u  (x::xs) =  fold1  f ( f  u  x)  xs;
```

```
fun reduce f  zero  [ ] = zero |
 reduce f  zero (x::tl) = f  x  (reduce f  zero tl);
```

fold1

>fun and x y:bool = if x then y else false;

val and = fn: bool → bool → bool

>fun fold1 and true [true];

val it = true: bool

*(*fold1 and true [true] = fold1 and (and true true) nil*
= fold1 and true nil
=true)*

```
fun fold1 f u nil = u |
   fold1 f u (x::xs) = fold1 f ( f u x) xs;
```


User defined Types in SML

datatype <name> = <constructor1> [of <type1>] |
<constructor2> [of <type2>] |…|
<constructorK> [of <typeK>];

Can define new sorts of data types this way and operations on them become discriminated by use of the constructor name.

Examples

> datatype Direction = N | S | E | W ;
datatype Direction **%enumeration**
con E: Direction
con N: Direction
con S: Direction
con W: Direction
>fun turn90 (N) = E | turn90 (E) = S |
turn90 (S) = W | turn90(W) = N;
val turn90 = fn :Direction → Direction

Examples

> datatype Length = Inches of real | Feet of real;
datatype Length
con Feet: real → Length
con Inches: real → Length
> fun circlearea(Inches r) = 3.14*r*r/144.0 |
circlearea(Feet f) = 3.14*f*f;
val circlearea = fn: Length → real
> circlearea (Inches 2.0); **can't just write (2.0) here!**
val it = 0.0872…2 :real

Parameterized polymorphic types

>datatype ‘a seq = nullseq | seq of ‘a * (‘a seq);

dataype ‘a seq

con nullseq → ‘a seq

*con seq: ‘a * ‘a seq → ‘a seq*

>val y = seq (2, nullseq);

val y = seq (2, nullseq) : int seq

>val z = seq (4, y);

val z = seq (4,y) : int seq

Examples- Recursive Types

>datatype ‘a bintree = leaf | node of ‘a * (‘a bintree) * (‘a bintree);

datatype ‘a bintree

con leaf : ‘a bintree

*con node: ‘a * (‘a bintree) * (‘a bintree)→`a bintree*

This tree stores data of type ‘a only at internal nodes.

Examples

>datatype ‘a tree = empty | leaf of ‘a | node of

‘a tree * ‘a tree;

datatype ‘a tree
con empty: ‘a tree
con leaf: ‘a → ‘a tree
con node : ‘a tree * ‘a tree → ‘a tree

Here the type signatures for leaf and node as constructors allow us to distinguish their uses from the previous datatype. This tree stores data only at its leaves.

Examples

>val x = node (1, node(2, leaf,leaf),leaf);

val x = node (1, node(2, leaf,leaf),leaf) : int bintree

>val y = empty;

val y = empty: ‘a tree

>val z = leaf 1;

val z = leaf 1: int tree

>val w = node (z,y);

val w = node (leaf 1, empty): int tree

**>datatype ‘a tree = empty |
leaf of ‘a |
node of ‘a tree * ‘a tree**

**>datatype ‘a bintree = leaf |
node of ‘a * (‘a bintree) * (‘a bintree);**

Examples

>val q = node (w, leaf);

ERROR: says node/2 expects int tree * int tree and

is given int tree * (‘z → ‘z tree) {type of leaf constructor in bintree}

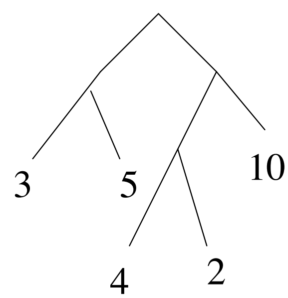
>val r = node(leaf 1, x);

ERROR: node /2 expects expects int tree * int tree and is given int tree * int bintree.

```
>datatype ‘a tree = empty |
   leaf of ‘a |
   node of ‘a tree * ‘a tree
```

```
>datatype ‘a bintree = leaf |
 node of ‘a * (‘a bintree) * (‘a bintree);
```


Functions on Datatypes



>fun countleaves (empty) = 0 |

countleaves (leaf (a)) = 1 |

countleaves (node (tree1, tree2)) =

countleaves (tree1) + countleaves (tree2);

val countleaves = fn: ‘a tree → int

>val tree = node (node (leaf(3),leaf(5)),

node((node(leaf(4), leaf(2)), leaf(10)));

val tree = …..: int tree

>countleaves (tree);

val it = 5: int

Function Closures

- *Closure* - a function and that part of its environment necessary for its evaluation
  - Needed for functions to be arguments

```
let val x =10
    fun f  y = x + y
in  f  3 end;
```

{λ y.x+y, x → 10 } closure of f

  - Can serve as a way of doing a kind of lazy evaluation of an unbounded argument

SML Sequences

A way to calculate effectively with a priori unbounded streams of data, by encapsulating a function.

```
datatype 'a seq =Nil | Cons of 'a * (unit → 'a seq)
fun head ( Cons (x,_) ) = x;
fun tail ( Cons (_, xf) ) = xf( );
```

Cons(x, fn() => <expr>) ; SML evaluates x but not the <expr>; this is prevented by the function abstraction. Cons here is a type constructor, NOT the same as the list constructor ::

Lisp calls these *streams.*

SML Sequences

>fun from k = Cons(k, fn()=>from (k+1));

val from = fn:int → int seq

> from 1;

val it = Cons(1,fn):int seq

> tail it;

val it = Cons(2,fn):int seq

>tail it;

val it = Cons(3,fn):int seq

Sequence is evaluated lazily; function abstraction hides an unbounded number of elements.

Example

>fun takeq (0, xq) = [ ] | fun takeq(n, Nil) = [ ] |
fun takeq(n, Cons(x,xf)) = x :: takeq(n-1, xf());
*val takeq=fn: int * ‘a seq → ‘a list*

>takeq(7, from 30);

[30,31,32,33,34,35,36] :int list

>takeq(3, from 5);

[5,6,7]: int list

How does this work??

Example

takeq(2, from 30) =

takeq(2, Cons (30, fn() => from (30+1))); =

30 :: (takeq(1, from (31))); =

30 :: (takeq(1,Cons(31, fn=>from(31+1)))); =

30 :: (31 :: (takeq(0,from (32))));

30 :: (31 :: (takeq(0,Cons(32,fn()=>from (32+1)))); =

30 :: (31 :: [ ]) = [30,31]

Notice that 32 is calculated but not used, so this calculation is not truly lazy.

```
fun takeq (0, xq) = [ ] | fun takeq(n, Nil) = [ ] |
fun takeq(n, Cons(x,xf)) = x :: takeq(n-1, xf());
```