Power Sets Revisited

Let’s compute power sets in ML.
We want a function pow that takes a list of values, viewed as a set, and which returns a list of lists. Each sublist will be one of the possible subsets of the original argument.

For example,

\[
\text{pow } [1,2] = \text{ [[1,2],[1],[2],[[]]]}
\]

We first define a version of cons in curried form:

\[
\text{fun cons h t = h::t;}
\]

\[
\text{val cons = fn :}
\]

\[
'a \to 'a \text{ list } \to 'a \text{ list}
\]
Now we define \texttt{pow}. We define the powerset of the empty list, $\texttt{[]}$, to be $\texttt{[][]}$. That is, the power set of the empty set is set that contains only the empty set.

For a non-empty list, consisting of $\texttt{h::t}$, we compute the power set of $\texttt{t}$, which we call \texttt{pset}. Then the power set for $\texttt{h::t}$ is just $\texttt{h}$ distributed through \texttt{pset} appended to \texttt{pset}.

We distribute $\texttt{h}$ through \texttt{pset} very elegantly: we just map the function $(\texttt{cons h})$ to \texttt{pset}. $(\texttt{cons h})$ adds $\texttt{h}$ to the head of any list it is given. Thus mapping $(\texttt{cons h})$ to \texttt{pset} adds $\texttt{h}$ to all lists in \texttt{pset}.
The complete definition is simply

```ml
fun pow [] = [[]]
| pow (h::t) = 
  let
    val pset = pow t
  in
    (map (cons h) pset) @ pset
  end;

val pow = 
  fn : 'a list -> 'a list list

Let's trace the computation of pow [1,2].

Here h = 1 and t = [2]. We need to compute pow [2].

Now h = 2 and t = [].

We know pow [] = [[]],
SO pow [2] =
(map (cons 2) [[]])@[[]] =
([[2]])@[[]] = [[2],[[]]]
Therefore \( \text{pow} \ [1,2] = \)
\[
(map \ (\text{cons} \ 1) \ [[2],[\;]])
\]
\[
@[[2],[\;]] =
\]
\[
[[1,2],[1]]@[[2],[\;]] =
\]
\[
[[1,2],[1],[2],[\;]]
\]
Composing Functions

We can define a composition function that composes two functions into one:

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

\[
\text{val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b}
\]

In curried form we have

\[
\text{fun comp } f \ g \ x = f(g(x));
\]

\[
\text{val comp = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b}
\]

For example,

\[
\text{fun sqr } x : \text{int} = x^2;
\]

\[
\text{val sqr = fn : int -> int}
\]

\[
\text{comp sqr sqr;}
\]

\[
\text{val it = fn : int -> int}
\]
comp sqr sqr 3;
val it = 81 : int

In SML \(\circ\) (lower-case O) is the infix composition operator.

Hence

\[\text{sqr} \circ \text{sqr} \equiv \text{comp} \ \text{sqr} \ \text{sqr}\]
Lambda Terms

ML needs a notation to write down unnamed (anonymous) functions, similar to the lambda expressions Scheme uses.

That notation is

``fn arg => body;
``

For example,

``val sqr = fn x : int => x*x;
val sqr = fn : int -> int
``

In fact the notation used to define functions,

``fun name arg = body;
``

is actually just an abbreviation for the more verbose

``val name = fn arg => body;
``
An anonymous function can be used wherever a function value is needed. For example,

map (fn x => [x]) [1,2,3];
val it = [[1],[2],[3]] : int list list

We can use patterns too:

(fn [] => []
  | (h::t) => h::h::t);
val it = fn : 'a list -> 'a list

(What does this function do?)
Polymorphism vs. Overloading

ML supports polymorphism.

A function may accept a polytype (a set of types) rather than a single fixed type.

In all cases, the same function definition is used. Details of the supplied type are irrelevant and may be ignored.

For example,

```ml
fun id x = x;
val id = fn : 'a -> 'a
fun toList x = [x];
val toList = fn : 'a -> 'a list
```
Overloading, as in C++ and Java, allows alternative definitions of the same method or operator, with selection based on type.

Thus in Java + may represent integer addition, floating point addition or string concatenation, even though these are really rather different operations.

In ML +, −, * and = are overloaded. When = is used (to test equality), ML deduces that an equality type is required. (Most, but not all, types can be compared for equality).

When ML decides an equality type is needed, it uses a type variable that begins with two tics rather than one.

```ml
fun eq(x, y) = (x=y);
val eq = fn : ''a * ''a -> bool
```
Defining New Types in ML

We can create new names for existing types (type abbreviations) using

\[
\text{type id } = \text{def;}
\]

For example,

\[
\text{type triple} = \text{int*real*string;}
\]

\[
\text{type triple} = \text{int * real * string}
\]

\[
\text{type rec1=}
\]

\[
\{a:\text{int}, b:\text{real}, c:\text{string}\};
\]

\[
\text{type rec1 =}
\]

\[
\{a:\text{int, b:real, c:string}\}
\]

\[
\text{type 'a triple3 } = 'a*'a*'a;
\]

\[
\text{type 'a triple3 } = 'a * 'a * 'a
\]

\[
\text{type intTriple } = \text{int triple3;}
\]

\[
\text{type intTriple } = \text{int triple3}
\]

These type definitions are essentially macro-like name substitutions.
The Datatype Mechanism

New types are defined using the \texttt{datatype} mechanism, which specifies new data value constructors. For example,

```plaintext
datatype color =
  red | blue | green;

datatype color =
  blue  | green | red
```

Pattern matching works on user-defined types using their constructors:

```plaintext
fun translate red = "rot"
| translate blue = "blau"
| translate green = "gruen";

val translate =
  fn : color -> string
```
fun jumble red = blue
  | jumble blue = green
  | jumble green = red;
val jumble = fn : color -> color
translate (jumble green);
val it = "rot" : string

**SML Examples**

Source code for most of the SML examples presented here may be found in
~cs538-1/public/sml/class.sml
Parameterized Constructors

The constructors used to define data types may be parameterized:

datatype money =
  none
  | coin of int
  | bill of int
  | iou of real * string;

datatype money =
  bill of int | coin of int
  | iou of real * string | none

Now expressions like \texttt{coin(25)} or \texttt{bill(5)} or \texttt{iou(10.25, "Lisa")} represent valid values of type \texttt{money}.  

We can also define values and functions of type `money`: 

```plaintext
val dime = coin(10);  
val dime = coin 10 : money  
val deadbeat =  
iou(25.00,"Homer Simpson");  
val deadbeat =  
iou (25.0,"Homer Simpson") : money  
fun amount(none) = 0.0  
| amount(coin(cents)) =  
  real(cents)/100.0  
| amount(bill(dollars)) =  
  real(dollars)  
| amount(iou(amt, _)) =  
  0.5*amt;  
val amount = fn : money -> real
```
Polymorphic Datatypes

A user-defined data type may be polymorphic. An excellent example is

datatype 'a option = none | some of 'a;

datatype 'a option = none | some of 'a

val zilch = none;

val zilch = none : 'a option

val mucho = some(10e10);

val mucho =
some 1000000000000.0 : real option

type studentInfo =
{name:string,
  ssNumber:int option};

type studentInfo = {name:string,
  ssNumber:int option}
val newStudent =  
{name="Mystery Man",  
 ssNumber=none} : studentInfo;

val newStudent =  
{name="Mystery Man",  
 ssNumber=none} : studentInfo
Datatypes may be Recursive

Recursive datatypes allow linked structures without explicit pointers.

datatype binTree =
  null
| leaf
| node of binTree * binTree;

datatype binTree =
  leaf | node of binTree * binTree
  | null

fun size(null) = 0
| size(leaf) = 1
| size(node(t1,t2)) = size(t1)+size(t2) + 1

val size = fn : binTree -> int
Recursive Datatypes may be Polymorphic

```ocaml
datatype 'a binTree =
  null
|  leaf of 'a
|  node of 'a binTree * 'a binTree

datatype 'a binTree =
  leaf of 'a |
  node of 'a binTree * 'a binTree |
  null

fun frontier(null) = []
  |  frontier(leaf(v)) = [v]
  |  frontier(node(t1,t2)) =
    frontier(t1) @ frontier(t2)

val frontier =
  fn : 'a binTree -> 'a list
```
We can model n-ary trees by using lists of subtrees:

```plaintext
datatype 'a Tree =
  null
| leaf of 'a
| node of 'a Tree list;
datatype 'a Tree = leaf of 'a | node of 'a Tree list | null

fun frontier(null) = []
| frontier(leaf(v)) = [v]
| frontier(node(h::t)) =
  frontier(h) @
  frontier(node(t))
| frontier(node([])) = []

val frontier = fn :
  'a Tree -> 'a list
```