Functional Programming With Lists

Amitabha Sanyal

Department of Computer Science and Engineering
IIT Bombay.
Powai, Mumbai - 400076

as@cse.iitb.ac.in

August 2014

Functional Programming with Lists

```
typedef enum {Nil_t,Cons_t}
                   Tagtype;
typedef struct L
{
  Tagtype tag;
    struct {
      int head;
      struct L* tail;
    };
} List;
```

```
List* Nil ()  {
    List* l = (List*) malloc(sizeof(List));
    l->tag = Nil_t;
    return(l);}

List* Cons (int hd, List* tl)  {
    List* l = (List*) malloc(sizeof(List));
    l->tag = Cons_t;
    l->head = hd;
    l->tail = tl;
    return(l);}
```

```
int main ()
  {
    int i;    List* l = Nil();
    i = 10;
    while (i > 0) l  = Cons (i--, l);}
```

Functional Programming with Lists

Lists in C

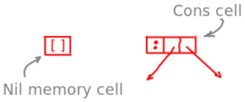
```
typedef enum {Nil_t,Cons_t}
                  Tagtype;
typedef struct L
{
  Tagtype tag;
  struct {
    int head;
    struct L* tail;};
} List;

List* Cons (int hd, List* tl)
{
 List* l = (List*) malloc(sizeof(l));
 l->tag = Cons_t;
 l->head = hd;
 l->tail = tl;
 return(l);}
```

Lists in Haskell

```
data List a = Nil | Cons a (List a)
```

- Defines a polymorphic type `List a`.
- `List` is a **type constructor**.
- `Nil` and `Cons` are **data constructors**. They also serve the role of tags.
- | is disjoint union.

```
data [a] = [] | a : [a]
```

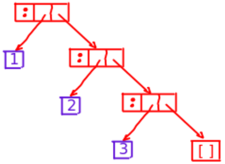


Functional Programming with Lists

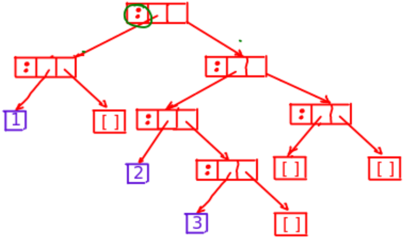
- `[]` is a shorthand for `Nil`.
- `[1,2,3]` is a shorthand for `1:(2:(3:[]))`.
- `1:(2:(3:[]))` is Haskell notation for `Cons 1 (Cons (2 (Cons 3 [])))`.

`[1,2,3]`

`['a', 'b', 'c'] :: [Char]`



`[[1], [2,3], []] :: [[Int]]`



Functional Programming with Lists

`length`

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum
```

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum  [] = 0
sum(x:xs) = x + sum xs

product
```

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum  [] = 0
sum(x:xs) = x + sum xs

product   [] = 1
product(x:xs) = x * product xs

[] ++ ys
```

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum   [] = 0
sum(x:xs) = x + sum xs

product    [] = 1
product(x:xs) = x * product xs

[] ++ ys    = ys
(x:xs)++ys = x:(xs ++ ys)

reverse
```

Functional Programming with Lists

```
length [] = 0                          map
length (x:xs) = 1 + length xs

sum  [] = 0
sum(x:xs) = x + sum xs

product   [] = 1
product(x:xs) = x * product xs

[] ++ ys    = ys
(x:xs)++ys = x:(xs ++ ys)

reverse  [] = []
reverse(x:xs) = reverse xs ++ x
```

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum  [] = 0
sum(x:xs) = x + sum xs

product   [] = 1
product(x:xs) = x * product xs

[] ++ ys   = ys
(x:xs)++ys = x:(xs ++ ys)

reverse  [] = []
reverse(x:xs) = reverse xs ++ x
```

```
map  f [] = []
map f (x : xs) = f x : map f xs

filter
```

Functional Programming with Lists

```
length [] = 0
length (x:xs) = 1 + length xs

sum  [] = 0
sum(x:xs) = x + sum xs

product   [] = 1
product(x:xs) = x * product xs

[] ++ ys    = ys
(x:xs)++ys = x:(xs ++ ys)

reverse  [] = []
reverse(x:xs) = reverse xs ++ x
```

```
map  f [] = []
map f (x : xs) = f x : map f xs

filter  p [] = []
filter p (x:xs)
    | p x = x:(filter p xs)
    | otherwise = (filter p xs)
```

The `foldr` function

`foldr` - the natural abstraction of list processing functions:

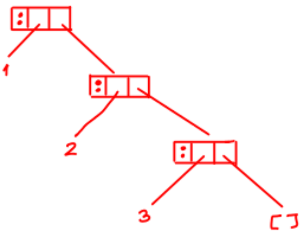
```
foldr f id [] = id
foldr f id (x:xs) = f x (foldr f id xs)
```

```
length l = foldr (\x y -> 1 + y) 0 l
l1 ++ l2 = foldr ...
reverse l = foldr ...
map f l = foldr ...
filter p l = foldr ...
```

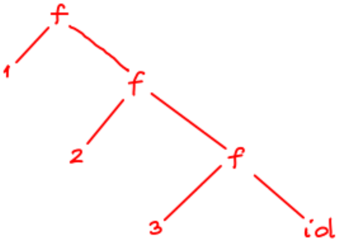
What does `foldr` do?

```
foldr f id [] = id
foldr f id (x:xs) = f x (foldr f id xs)
```

`[1,2,3]`



`foldr f id [1,2,3]`



Other List Processing Functions

```
xs !! n  | n < 0 = error "negative index"
[] !! n          = error "index too large"
(x:xs)  !! 0     = x
(x:xs)  !! n     = xs !! (n-1)

take 0 _  =  []
take _ [] =  []
take (n + 1) (x:xs) = x : take n xs

zip [] l2 = []
zip l1 [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

takeWhile p [] = []
takeWhile p (x:xs) = if (p x) then x : takewhile p xs else []

dropWhile p [] = []
dropWhile p (x:xs) = if (p x) then dropWhile p xs else x:xs
```

List Comprehensions

```
We wanted a function dropWhile' which satisfied
dropWhile' p l = (dropWhile p l, l)

Now we wanted to express dropWhile' as a foldr, i.e.
dropWhile' p l = foldr g id l

Let us calculate g and id.
id = dropWhile' p [] = (dropWhile p [], [])

Now dropWhile' p (x:xs)
  = (dropWhile p (x:xs), (x:xs))
  = (if (p x) then dropWhile p xs else x:xs, x:xs)
  = if (p x) then (dropWhile p xs, x:xs) else (x:xs, x:xs)
  = if (p x) then (dropWhile p xs), x:xs) else (x:xs, x:xs)
  = if (p x) then (d, x:l) else (x:xs, x:xs)
       where (d, l) = dropWhile' p xs  -- and l = xs

Extract the g
g x (d,l) = if (p x) then (d, x:l) else (x:l, x:l)
```

List Comprehensions

```
[x * x | x <- [1,2,3,4,5]] => [1,4,9,16,25]
[x * x | x <- [1,2,3,4,5], even x] => [4,16]
[x + y | | x <- [1,2,3], y <- [6,7]] => [7,8,8,9,9,10]
```

```
qsort [] = []
qsort (x:xs) = qsort lows ++ [x] ++ qsort highs
     where lows =  [y | y <- xs, y <= x]
           highs = [y | y <- xs, y > x]
```

```
fib = 0:1:[x + y | (x,y) <- zip fib (tail fib)]
```

```
fib = 0:1:[x + y | x <- fib, y <- (tail fib)]
```

does not work. Why?

List Comprehension

The eight queens problem

```
queens 0 = [[]]
queens n = [board ++ [pos] | board <- queens (n-1),
                             pos <- [1..8],
                             safeconfig board pos]
     where safeconfig board pos = all (safepos (n,pos))
                                      (zip [1..n-1] board)
           safepos (n1, pos1) (n, pos) = pos /= pos1 &&
                                         abs (n-n1) /= abs (pos-pos1)
```

Reasoning about Functional Programs

Universal property of foldr:

```
e [] = id                ⇔   e = foldr f id
e (x:xs) = f x (e xs)
```

<= `foldr f id` is a solution of the equations on the left

>= `foldr f id` is the only solution of the equations on the left

Reasoning about Functional Programs

Universal property of foldr:

```
e [] = id                ⇔   e = foldr f id
e (x:xs) = f x (e xs)
```

<= `foldr f id` is a solution of the equations on the left

>= `foldr f id` is the only solution of the equations on the left

One can prove many interesting results using this property

1. `(+ 1).sum = foldr (+) 1`
2. if `h w = v` and `h (g x y) = f x (h y)`, then `h.(foldr g w) = foldr f v`.
3. `map s.map t = map (s.t)`
4. `map s.concat = concat.map (map s)`

Reasoning about Lists

`inits`: Finds all initial segments of a list.

```
inits [] = [[]]
inits (x:xs) = [] : (map (x:) inits xs)
```

`tails`: Finds all tail segments of a list

```
tails [] = [[]]
tails (x:xs) = (x:head l):l
   where l = tails xs
```

Can you express `heads` and `tails` using `foldr`?

Reasoning about Functional Programs

scanl/scanr: Accumulating foldl/foldr. *n* times the the time taken to apply f on a list element.

```
scanl f id l = map (foldl f id ) (inits l)
scanr f id l = map (foldr f id ) (tails l)
```

foldr1: Fold without identity element

```
foldr1 f (x:xs) = if null xs then x else f x (foldr1 f xs)
```

Reasoning about Functional Programs

Bookkeeping law: If `f` is associative with identity `a`. Then:

```
foldr f a . concat = foldr f a . map (foldr f a)
```

Generalized Horner's rule:

$$1 + x_0 + x_0 * x_1 + x_0 * x_1 * x_2 + x_0 * x_1 * x_2 * x_3 \qquad \text{– } \mathit{foldr1} + \ . \ \mathit{scanl} * 1$$
$$= 1 + x_0 * (1 + x_1 * (1 + x_2 * (1 + x_3))) \qquad \text{– } \mathit{foldr}\ f\ 1\ \mathit{where}\ f\ x\ y = 1 + x * y$$

Under what conditions is:

$\mathtt{foldr1} \oplus . \mathtt{scanl} \otimes \mathtt{id}$ the same as $\mathtt{foldr} \odot \mathtt{id}$

where $\mathtt{x} \odot \mathtt{y} = \mathtt{id} \oplus (\mathtt{x} \otimes \mathtt{y})$?

Answer:

1. `id` should be the identity element of $\otimes$.
2. $\otimes$ should distribute over $\oplus$.
3. $\otimes$ should be associative.

The maximum segment sum problem

Given a sequence of integers, find the maximum of the sum of all (contiguous) segments.

Example:

```
mss [-1, 2, -3, 5, -2, 1, 3, -2, -2, -3, 6] = 7
```

An obviously correct but inefficient definition of `mss` :

```
mss = maximum . map sum . segs
segs = concat . map inits . tails
```

1. `tails` – $O(n)$
2. `map inits` – $O(n^3)$, assuming $O(n)$ sublists, each of length $O(n)$
3. `concat` – $O(n^2)$, assuming $O(n)$ sublists, each of length $O(n)$.
4. `map sum` – $O(n^3)$, assuming $O(n^2)$ sublists, each of length $O(n)$.

Reasoning about Functional Programs

```
mss = maximum . map sum . concat . map inits . tails
       {map f . concat = concat . map (map f)}
    = maximum . concat . map (map sum) . map inits . tails
       {map f . map g = map (f . g)}
    = maximum . concat . map (map sum . inits) . tails
       {maximum = foldr max -infinity}
    = maximum . map maximum . map (map sum . inits) . tails
       {map f . map g = map (f . g)}
    = maximum . map (maximum . map sum . inits) . tails
       {map sum . inits = scanl (+) 0}
    = maximum . map (maximum . scanl (+) 0) . tails
       {maximum = foldr1 max}
```

{ 0 is id for $+$}

{ $+$ is associative}

{ $+$ distributes over max}

```
    = maximum . map (foldr f 0) . tails
        where f x y = 0 `max` (x+y)
    = maximum . scanr f 0
```

A Sudoku solver

As an example of

- List processing

- Backtracking in lazy languages

Reference: FUNCTIONAL PEARL - A program to solve Sudoku, RICHARD BIRD

www.cs.tufts.edu/~nr/comp150fp/archive/richard-bird/sudoku.pdf

The Board

row

column

box

```
board1 = [['2', '.', '.', '.', '.', '1', '.', '3', '8'],
          ['.', '.', '.', '.', '.', '.', '.', '.', '5'],
          ['.', '7', '.', '.', '.', '6', '.', '.', '.'],
          ['.', '.', '.', '.', '.', '.', '.', '1', '3'],
          ['.', '9', '8', '1', '.', '.', '2', '5', '7'],
          ['3', '1', '.', '.', '.', '.', '8', '.', '.'],
          ['9', '.', '.', '8', '.', '.', '.', '2', '.'],
          ['.', '5', '.', '.', '6', '9', '7', '8', '4'],
          ['4', '.', '.', '2', '5', '.', '.', '.', '.']]
```

```
type Matrix a = [[a]]
type Board = Matrix Char
```

Characterizing a correct solution

Some constants

```
boxsize = 3:: Int
allvals = "123456789"
blank c = c == '.'
```

A Board is correct, if each row, each column and each box is free of duplicates.

```
correct :: Board -> Bool

correct b = all nodups (rows b) &&
            all nodups (cols b) &&
            all nodups (boxes b)

nodups [] = True
nodups (x:xs) = notElem x xs && nodups xs
```

Characterizing a correct solution

```
rows = id
```

cols makes rows of columns:

```
cols [] = replicate 9 []
cols (r:rs) = zipWith (:) r (cols rs)
```

boxes makes rows of boxes:

```
board1 = [['2', '.', '.', '.', '.', '1', '.', '3', '8'],
          ['.', '.', '.', '.', '.', '.', '.', '.', '5'],
          ['.', '7', '.', '.', '.', '6', '.', '.', '.'],
          ['.', '.', '.', '.', '.', '.', '.', '1', '3'],
          ['.', '9', '8', '1', '.', '.', '2', '5', '7'],
          ['3', '1', '.', '.', '.', '.', '8', '.', '.'],
          ['9', '.', '.', '8', '.', '.', '.', '2', '.'],
          ['.', '5', '.', '.', '6', '9', '7', '8', '4'],
          ['4', '.', '.', '2', '5', '.', '.', '.', '.']]
```

Characterizing a correct solution

```
boxes = map unchop . unchop . map cols . chop . map chop

chop :: [a] -> [[a]]
chop = chopBy boxsize
  where chopBy bsize [] = []
        chopBy bsize l = (take bsize l) :
                         (chopBy bsize (drop  bsize l))
unchop = concat
```

Notice that rows, cols or boxes done twice gives identity.

```
rows . rows = id
cols . cols = id
boxes . boxes = id
```

Choices

A Choice is a list of characters, that represent the choices for a cell

- Intially, the choices for a blank cell are all possible characters, and
  the choices for a non-blank cell is the only character in the cell.

```
type Choices = [Char]

initialChoices :: Board -> Matrix Choices
initialChoices = map (map fillin)
  where fillin initialChar = if blank initialChar then allvals
                                                    else [initialChar]
```

From a Matrix of Choices, we want to generate all possible boards.

How does one do that?

Easier problem: From a list of choices, how does one generate all possible list?

Choices

```
cp :: [[a]] -> [[a]]  -- cp for cartesian product
cp [] = [[]]
cp (x:xs) = [h:t | h <- x, t <- (cp xs)]
```

How can one use cp, to calculate the matrix cartesian product, mcp?

```
mcp :: Matrix [a] -> [Matrix a]
```

Surprisingly, mcp is easy to define using cp?

```
mcp = cp (map cp)
```

map cp converts a Matrix of choices into

[list of possible first rows
list of possible second rows
...
list of possible ninth rows]

cp converts it into possible matrix of Boards

Choices

A Sudoku solver takes a board and returns all possible completions of the board. Returns [] if there are none.

```
sudokusolver1 :: Board -> [Board]

--sudokusolver - first attempt

sudokusolver1  =  filter correct . mcp . initialChoices
```

Pruning

We would like to do pruning of the following form:

24	2	34	12
34	234	134	13
124	23	13	4
14	123	123	3

4	2	34	1
34	34	134	1
12	3	13	4
14	1	12	3

This is one time pruning.

Given a row, column or a box, we collect all the fixed choices and remove these from the non-fixed choices.

```
fixed :: [Choices] -> Choices   -- fixed identifies fixed choices
fixed  = concat . filter single
    where single [_] = True
          single  _   = False
```

Pruning

pruneList takes a list of choices and prunes it into a list of choices:

```
pruneList :: [Choices] -> [Choices]

pruneList css = map (remove (fixed css)) css
  where remove fs cs = if single cs then cs else delete fs cs
        delete fs cs = filter (\c -> not (c `elem` fs)) cs
```

Now pruneMatrix prunes each row, each column and each box using pruneList

24	2	34	12
34	234	134	13
124	23	13	4
14	123	123	3

24	2	34	234

4	2	34	12
34	34	134	13
124	23	13	4
14	123	123	3

Pruning

The rows pruning can be done by

```
rows . map pruneList . rows
```

Similarly for pruning by columns and boxes. We therefore abstract:

```
pruneBy f = f . map pruneList . f
pruneMatrix = pruneBy boxes. pruneBy cols . pruneBy rows

sudokusolver2 :: Board -> [Board]
sudokusolver2  =  filter correct.mcp .pruneMatrix.initialChoices
```

plug in your own
pruning strategy here

Expand → Prune → Expand → Prune

24	2	4	12
34	234	134	13
124	23	13	4
14	123	123	3

← Expand

24	2	34	12
34	234	134	13
124	23	13	4
14	123	123	3

Expand →

24	2	3	12
34	234	134	13
124	23	13	4
14	123	123	3

↓ Prune

	2	4	1
34	34	13	1
12	3	1	4
14	1	12	3

Blocked

Prune ↓

4	2	3	1
34	34	14	1
12	3	1	4
14	1	12	3

Blocked

Expand → Prune → Expand → Prune

Take a Choice Matrix that has a cell with at least two (say x) choices, and generate x Choice Matrices with fixed choice for this cell.

```
expand :: Matrix Choices -> [Matrix Choices]
```

Sometimes a Choice Matrix can be blocked. Conditions are

1. No choices for a cell,
2. Same fixed choices in more than one cells in row, col or box.

We shall discard blocked matrices during expansion and pruning.

Expand → Prune → Expand → Prune

```
blocked :: Matrix Choices -> Bool
blocked cm = void cm || not (safe cm)

void :: Matrix Choices -> Bool
void cm  = any (any null) cm

safe :: Matrix Choices -> Bool
safe cm = all (nodups . fixed) (rows cm) &&
          all (nodups . fixed) (cols cm) &&
          all (nodups . fixed) (boxes cm)
```

Expand → Prune → Expand → Prune

To expand, we select the first cell that has the minimum number of choices amongst all cell which have more than one choices.

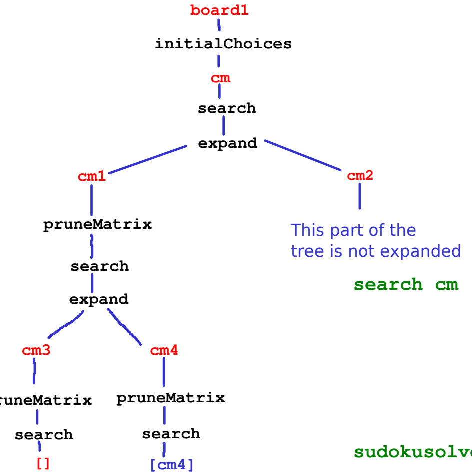
```
minchoice = minimum . filter ( > 1) . concat . map (map length)
```

A choice list is a candidate for expansion if its length is the same as the minimum. We pick the first candidate for-- expansion. This goes as follows:

```
expand cm = [rows1 ++ [row1 ++ [c] : row2] ++ rows2 | c <- cs]
  where (rows1, row:rows2) = break (any isCandidate) cm
        (row1, cs:row2) = break isCandidate row
        isCandidate cs = (length cs == n)
        n = minchoice cm
```

The Final Solution

```
search cm | blocked cm = []
          | all (all single) cm = [cm]
          | otherwise = (concat .
                         map (search . pruneMatrix) .
                         expand) cm

sudokusolver3 =  map (map head) .  head . search . initialChoices
```




```
search cm | blocked cm = []
          | all (all single) cm = [cm]
          | otherwise = (concat .
                           map (search . pruneMatrix) .
                           expand) cm

sudokusolver3 =  map (map head) .
                 head . search .
                 initialChoices
```