Functional Programming With Trees

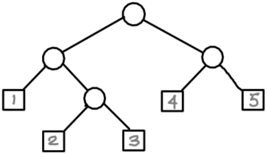
Amitabha Sanyal

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

as@cse.iitb.ac.in

August 27 2014

Binary Trees



Binary trees are defined as:

```
data Btree a = Leaf a | Fork (Btree a) (Btree a)
```

The tree above is represented as:

```
bt1 :: Btree Int
bt1 = Fork (Fork (Leaf 1) (Fork (Leaf 2) (Leaf 3)))
           (Fork (Leaf 4) (Leaf 5))
```

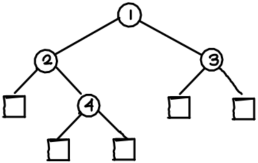
Binary Trees

Functions on binary trees:

```
size (Leaf x) = 1
size (Fork xt yt) = 1 + size xt + size yt

mirror (Leaf x) = Leaf x
mirror (Fork xt yt) = Fork (mirror yt) (mirror xt)

flatten (Leaf x) = [x]
flatten (Fork xt yt) = flatten xt ++ flatten yt

flatten t = flatten' t []
  where flatten' (Leaf x) l = x : l
        flatten' (Fork xt yt) l = flatten' xt (flatten' yt l)
```

Binary Search Trees

Used for representing sets:



```
data (Ord a) => Stree a = Null | Fork a (Stree a) (Stree a)
  deriving Show
```

Property of a binary serch tree:

If `Fork v xt yt` is a binary search tree, then for all `x`,
`member x xt` $\Rightarrow$ `x` $<$ `v`, and `member x yt` $\Rightarrow$ `x` $>$ `v`

Functions on Binary Search Trees

Functions on binary search trees:

```
flatten Null = []
flatten (Fork x xt yt)    = flatten xt ++ [x] ++ flatten yt

insert x Null = Fork x Null Null
insert x t@(Fork y xt yt) | x < y = Fork y (insert x xt) yt
                          | x == y = t
                          | x > y = Fork y xt (insert x yt)
member x Null = False
member x (Fork y xt yt) | x < y = member x xt
                          | x == y = True
                          | x > y = member x yt

delete x Null = Null
delete x (Fork y xt yt)  | x < y = Fork y (delete x xt) yt
                         | x == y = join xt yt
                         | x > y = Fork y xt (delete x yt)
```

Functions on Binary Search Trees

How should join be defined?

- The value at any node in `xt` is less than the value of all nodes in `yt`.
- One possible way to join:

- Results in skewed trees, not good for search.

Functions on Binary Search Trees

What property should `join` satisfy?

```
flatten(join xt  yt) = flatten  xt ++ flatten  yt
```

Now we have two cases:

1. `yt` is `Null`. Then

   ```
   flatten(join xt Null) = flatten xt ++ flatten Null
                         = flatten  xt ++ []
                         = flatten  xt
   ```

   Cancelling `flatten` on both sides we have:
   `join xt Null = xt`

Functions on Binary Search Trees

2. yt is not Null.

   We then have:

```
flatten(join xt  yt) = flatten  xt ++ flatten  yt
                     = flatten  xt ++ [head (flatten  yt)]
                                   ++ tail (flatten  yt)
```

   In this case, we guess the following:

   `join xt yt = Fork (ht yt) xt (tt yt)`.

   - The value at the root is some function `ht` of `yt` (only).
   - The right subtree is also some function `tt` of `yt` .

```
flatten(join xt yt) = flatten (Fork (ht yt) xt (tt yt))
                    = flatten xt ++ [(ht yt)] ++ flatten (tt yt)
```

   This gives:

```
ht yt = head (flatten  yt)
flatten(tt yt) = tail (flatten  yt)
```

Functions on Binary Search Trees

From this we can synthesize the following definitions:

```
ht (Fork v Null ytr) = head (flatten (Fork v Null ytr))
                     = head (flatten Null ++ [v] ++ flatten ytr)
                     = head ([v] ++ flatten ytr)
                     = v


ht (Fork v ytl ytr)  = head (flatten (Fork v ytl ytr))
                     = head (flatten ytl ++ [v] ++ flatten ytr)
                     = head (flatten ytl)
                     = ht ytl
```

Functions on Binary Search Trees

```
flatten (tt (Fork v Null ytr)) = tail (flatten  (Fork v Null ytr))
                               = tail (flatten Null ++ [v] ++ flatten ytr)
                               = tail ([v] ++ flatten ytr)
                               = flatten ytr
```

Thus `tt (Fork v Null ytr) = ytr`. Further

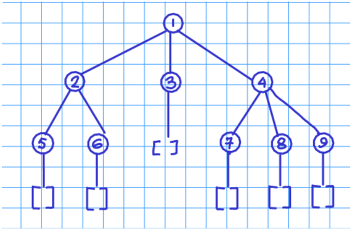
```
flatten (tt (Fork v ytl ytr)) = tail (flatten  (Fork v ytl ytr))
                              = tail (flatten ytl ++ [v] ++ flatten ytr)
                              = tail (flatten ytl) ++ [v] ++ flatten ytr
                              = flatten (tt ytl) ++ [v] ++ flatten ytr
                              = flatten (Fork v (tt ytl) ytr)
```

Thus `tt (Fork v ytl ytr) = Fork v (tt ytl) ytr`.

General (n-ary) trees

General Trees

```
data Gtree a = Node a [Gtree a]

gt1 :: Gtree Int
gt1 = Node 1 [Node 2 [Node 5 [], Node 6 []],
              Node 3 [],
              Node 4 [Node 7 [], Node 8 [], Node 9 []]]
```



Min-max with alpha beta pruning

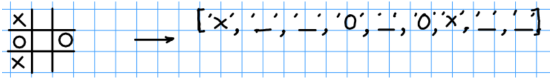
Consider representing the moves of a game with a function `moves`

```
moves :: Position -> [Position]
```

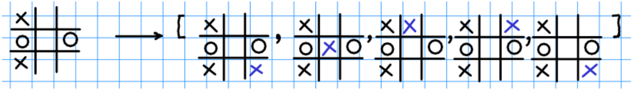
We do not define `moves` or `position` any further.

Example: Tic-Tac-Toe

Represent a `Position` as a list:



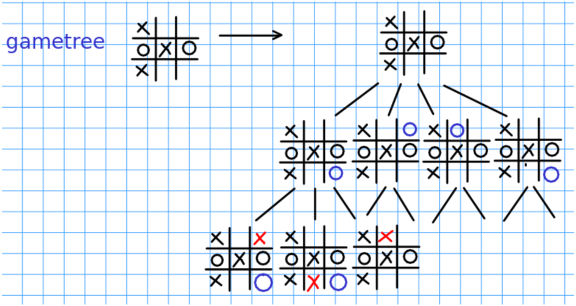
And `moves` as a function:



Min-max with alpha beta pruning

Given moves we can define the evolution of a game starting from a position pos:

```
reptree f initial = Node initial (map (reptree f) (f initial))
gametree pos = reptree moves pos
```



gametree can return a very large, or for some games, even an infinite tree.

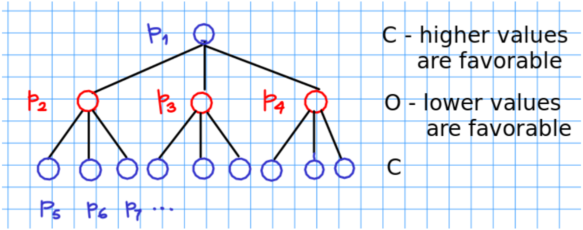
Min-max with alpha beta pruning

```
static :: Position -> Value
```

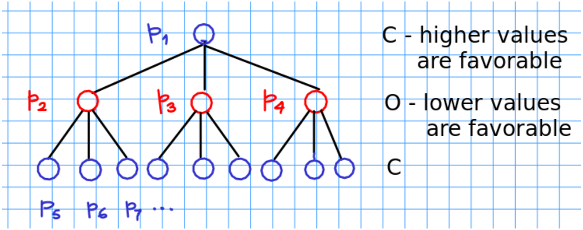
Gives a static value to a board position without evaluating the game.

```
data Who = C | O
dynamic :: Who -> Gtree Position -> Value
```

Gives a value to the root of a `gametree` by traversing the tree for a few levels.

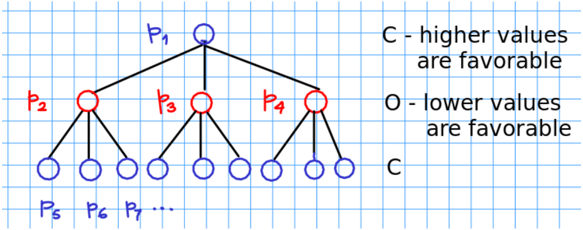


Min-max with alpha beta pruning



- Question: What is the dynamic value of p_1?
  Answer: The maximum of the dynamic values of p_2, p_3, p_4.
- Question: What is the dynamic value of p_2?
  Answer: The minimum of the dynamic values of p_5, p_6, p_7.
- Question: What is the dynamic value of p_6?
  Answer: The static value p_6, assuming no lookahead.

Min-max with alpha beta pruning




```
dynamic C (Node pos []) = static pos
dynamic C (Node pos l) = maximum (map (dynamic O) l)
dynamic O (Node pos []) = static pos
dynamic O (Node pos l) = minimum (map (dynamic C) l)

evaluate = dynamic C . gametree
```

- If `gametree` returns an infinite tree, then `evaluate` will not terminate.
- How do we add a termination condition without disturbing the existing code?

Min-max with alpha beta pruning

```
evaluate = dynamic C . prune 5 . gametree
```

Min-max with alpha beta pruning

```
evaluate = dynamic C . prune 5 . gametree

prune 1 (Node pos l) = Node pos []
prune n t@(Node pos []) = t
prune n (Node pos l) = Node pos (map (prune (n - 1)) l)
```

Min-max with alpha beta pruning

```
evaluate = dynamic C . prune 5 . gametree

prune 1 (Node pos l) = Node pos []
prune n t@(Node pos []) = t
prune n (Node pos l) = Node pos (map (prune (n - 1)) l)
```

- Traversal over the gametree for its evaluation and termination of the evaluation process are separate concerns.
- These could be kept apart due to laziness.

Min-max with alpha beta pruning

```
evaluate = dynamic C . prune 5 . gametree

prune 1 (Node pos l) = Node pos []
prune n t@(Node pos []) = t
prune n (Node pos l) = Node pos (map (prune (n - 1)) l)
```

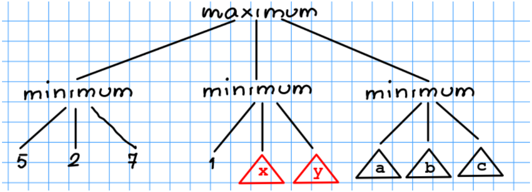
- Traversal over the gametree for its evaluation and termination of the evaluation process are separate concerns.
- These could be kept apart due to laziness.

`test.generate` has different interpretations in lazy and eager languages:

- **Eager languages:** Generate and then test.
- **Lazy languages:** Generate only as much as to pass test.

Min-max with alpha beta pruning

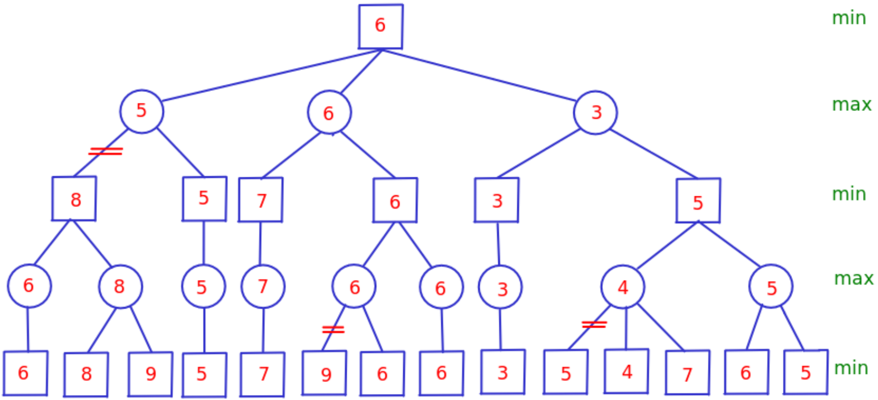
Alpha-Beta pruning:



$$
\begin{aligned}
& \mathit{maximum}\ (\mathit{minimum}\ 5\ 2\ 7)\ (\mathit{minimum}\ 1\ \mathtt{x}\ \mathtt{y})\ (\mathit{minimum}\ a\ b\ c) \\
= \ & \mathit{maximum}\ 2\ (\mathit{minimum}\ 1\ \mathtt{x}\ \mathtt{y})\ (\mathit{minimum}\ a\ b\ c) \\
= \ & \mathit{maximum}\ 2\ n\ (\mathit{minimum}\ a\ b\ c), \text{ where } n \leq 1 \\
= \ & \mathit{maximum}\ 2\ (\mathit{minimum}\ a\ b\ c)
\end{aligned}
$$

Therefore $\mathtt{x}$ and $\mathtt{y}$ need not be evaluated.

Min-max with alpha beta pruning



Min-max with alpha beta pruning

Can we modify the computation of `dynamic C` to incorporate this modification?

```
dynamic C (Node pos []) = static pos
dynamic C (Node pos l) = min maxInt (dynamic C (Node pos l))
                       = alpha (Node pos l) maxInt,
    where alpha (Node pos l) potmin =  min potmin (dynamic C (Node pos l))
```

Min-max with alpha beta pruning

Can we modify the computation of `dynamic C` to incorporate this modification?

```
dynamic C (Node pos []) = static pos
dynamic C (Node pos l) = min maxInt (dynamic C (Node pos l))
                       = alpha (Node pos l) maxInt,
    where alpha (Node pos l) potmin =  min potmin (dynamic C (Node pos l))
```

In fact, we can combine the two clauses of `dynamic C` to get:

```
dynamic C (Node pos l) = alpha (Node pos l) maxInt
    where alpha (Node pos []) potmin = static pos
          alpha (Node pos l) potmin =  min potmin (dynamic C (Node pos l))
```

Can we calculate `min potmin (dynamic C (Node pos l))` efficiently?

```
      min potmin (dynamic C (Node pos l))
   =  min potmin (maximum (map (dynamic O) l)))
   =  min potmin (maximum xs), where xs = map (dynamic O) l
   =  ((min potmin).maximum) xs
   =  ((min potmin).(foldr max minInt)) xs
```

Min-max with alpha beta pruning

Can we express `(min potmin).(foldr max minInt)` as a `foldr`?

Exercise: Show that

```
(min potmin).(foldr max minInt) = foldr g id
   where  id = min potmin minInt = minInt, and
          g a potmax = max (min potmin a) potmax -- Note the invariant
                                                 --  potmax <= potmin
                     = min (max potmin potmax) (max a potmax)
```

Min-max with alpha beta pruning

Can we express `(min potmin).(foldr max minInt)` as a `foldr`?

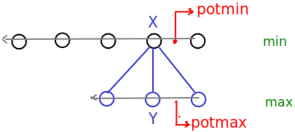
Exercise: Show that

```
(min potmin).(foldr max minInt) = foldr g id
   where  id = min potmin minInt = minInt, and
          g a potmax = max (min potmin a) potmax -- Note the invariant
                                                 --  potmax <= potmin
                     = min (max potmin potmax) (max a potmax)
```

We can further rewrite this to

```
g a potmax | potmax >= potmin = potmax -- The great alpha pruning
           | potmax < potmin  = min potmin (max a potmax)
                              = min potmin (max potmax a)
```

Min-max with alpha beta pruning



- Computation of Y is important for the subsequent potmin, if it can possibly lower it from the current value. This will happen if X (which is computed from Y) is lower than potmin.
- The value of X is $(\text{potmax} + \Delta), \Delta \geq 0$.
- If potmax $\geq$ potmin, then X is no less than potmin. In this case, not important to evaluate Y.

Min-max with alpha beta pruning

The development so far is that

```
alpha (Node pos l) potmin =  foldr g minInt  (map (dynamic 0) l)
where
   g a potmax | potmax >= potmin = potmax -- The great alpha pruning
              | potmax < potmin  = min potmin (max potmax a)
```

Min-max with alpha beta pruning

The development so far is that

```
alpha (Node pos l) potmin =  foldr g minInt  (map (dynamic 0) l)
where
   g a potmax | potmax >= potmin = potmax -- The great alpha pruning
              | potmax < potmin  = min potmin (max potmax a)
```

Now, can we express `foldr g minInt.(map (dynamic 0))` as a `foldr`

Exercise: Show that

```
 foldr g minInt .(map (dynamic 0)) = foldr h minInt
   where  h a potmax = g (dynamic 0 a) potmax
```

Min-max with alpha beta pruning

Substituting for the value of `g` in the definition of `h` we get:

```
foldr g minInt . (map (dynamic 0)) = foldr h minInt
  where  h a potmax | potmax >= potmin = potmax
                    | potmax < potmin  = min potmin (max potmax (dynamic 0 a))
```

We can do a dual development for `(max potmax (dynamic 0 a))`The resulting function is called `beta`:

Min-max with alpha beta pruning

We obtain:

```
dynamic C (Node pos []) = alpha (Node pos l) maxInt
       where alpha (Node pos []) potmin = static pos
             alpha (Node pos l) potmin = foldr h minInt l
               where h a potmax | potmax >= potmin = potmax -- alpha pruning
                     h (Node pos' l) potmax = min potmin
                                                 (beta (Node pos' l) potmax)
             beta (Node pos []) potmax = static pos
             beta (Node pos l) potmax = foldr g maxInt l
               where g a potmin | potmin <= potmax = potmin -- beta pruning
                     g (Node pos' l) potmin = max potmax
                                                 (alpha (Node pos' l) potmin)
```