Concept of Programming Languages (CS320)
Lecture 5

By Zhiqiang Ren (Alex)
aren@cs.bu.edu
Content

- Functional vs. Imperative Data Structures
- Terminology of Data Structure
- Tree Operations
- Option Type
Functional vs. Imperative Data Structures

- Functional Programming contributes greatly to modularity.
  - Easy to write, debug.
  - A collection of modules that can be reused to reduce future programming costs.
- Functional data structures are more difficult to design and implement than imperative ones
  - No destructive update (i.e., assignment)
  - Old version of the data structure survives the update.
    - Persistent Data Structure
    - Ephemeral Data Structure
Terminology of data structure

- An *abstract data type* (a type and a collection of functions on that type) – abstraction
  - .sats files
- A *concrete realization* of an abstract data type – implementation
- An *instance of a data type* (a particular list or tree) – an object, a stack, a queue.
- A *unique identity* that is invariant under updates – persistent identity
Tree

- Binary Tree

datatype itree =
  | nil
  | cons of (itree, int, itree)
Tree Operations

- **Size**
  
  ```
  fun itree_size (t: itree): int =
  case+ t of
    | nil () => 0
    | cons (tl, _, tr) => itree_size (tl) + itree_size (tr) + 1
  ```

- **Height**
  
  ```
  fun itree_height(t: itree): int =
  case+ t of
    | nil () => 0
    | cons (tl, _, tr) => max(itree_height(t), itree_height(t)) + 1
  ```
Balanced Tree

- The depth of the two subtrees of every node differ by 1 or less
- Height = \([\log_2(\text{size})]+1\)
Tree Operations

- Rotate (promote balance)
Option Type

- Another way of indicating error.

datatype `option0` (a: `t@ype`) =
| `option0_none` (a)
| `option0_some` (a) of a

fun `isAVL` (t: `itree`): `option0` int =
case+ t of
| nil () => `option0_some` (0)
| cons (tl, _, tr) =>
  case+ (`isAVL` (tl), `isAVL` (tr)) of
    | (`option0_some` (hl), `option0_some` (hr)) =>
      if abs (hl - hr) <= 1 then `option0_some` (max (hl, hr) + 1)
      else `option0_none` ()
    | (_, _) => `option0_none` ()
Tree Operation

- Tree Traversal (preorder, inorder, postorder)

```haskell
fun preorder (t: itree): list0 int =
case+ t of
| nil () => list0_nil ()
| cons (tl, v, tr) => let
  val ll = preorder (tl)
  val lr = preorder (tr)
  val v_ll = list0_cons (v, ll)
  val v_ll_lr = list0_append (v_ll, lr)
in
  v_ll_lr
end
```
Tree Operation

- Tree Traversal (another version)

```ocaml
fun preorder2 (t: itree): list0 int = let
    fun preorder2_impl (t: itree, xs: list0 int): list0 int =
        case+ t of
        | nil () => xs
        | cons (tl, v, tr) => let
            val lr_xs = preorder2_impl (tr, xs)
            val ll_lr_xs = preorder2_impl (tl, lr_xs)
        in
            list0_cons (v, ll_lr_xs)
        end
    in
        preorder2_impl (t, list0_nil ())
    end
```
List to Tree

• Construct a balanced tree

fun balpre (xs: list0 int): itree =
case+ xs of
| list0_nil () => nil
| list0_cons (x, xs') => let
  val len = list0_length (xs')
  val ll = list0_take_exn (xs', len / 2)
  val lr = list0_drop_exn (xs', len / 2)
  val tl = balpre (ll)
  val tr = balpre (lr)
in
  cons (tl, x, tr)
end
Binary Search Tree

- isBST
Binary Search Tree

- insert (leaf insertion)