15-150 Fall 2019

Tuesday, 22 October

Stephen Brookes
Today

A case study in *modular programming*

• an abstract type of *dictionaries*

- signatures
- structures
- functors
- information hiding
dictionaries

Earlier: **sorting** with a *comparison function*

Now: revisit, using **modular** design

- A signature for *dictionaries*
  - a dictionary is a collection of *entries*, sorted by *keys* drawn from an *ordered type*

- A structure *implementing* dictionaries
  - *binary search trees*, sorted by *key*
signatures

signature ORDERED =
  sig
    type t
    val compare : t * t -> order
  end

signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : Key.t * 'a -> 'a dict -> 'a dict
    val lookup : Key.t -> 'a dict -> 'a option
    val trav : 'a dict -> (Key.t * 'a) list
  end
specifications

• An ordered type is a structure $K : \text{ORDERED}$ for which $K.\text{compare}$ is a comparison function on type $K.t$

• A dictionary implementation is a structure $S : \text{DICT}$ for which
  
  | S.empty  | - the empty dictionary |
  | S.insert  | - insert a key-value entry |
  | S.lookup  | - find the entry with a given key |
  | S.trav    | - list the entries in key order |

satisfy “the usual equations”
specifications

- In class we won’t have time to elaborate on the “usual equations” or give proof details
- Here is a representative sample…

(a) \text{lookup } k \text{ empty } = \text{NONE}
(b) \text{lookup } k \text{ (insert } (k', v') \text{ } D) = \text{SOME } v'
  \quad \text{if } \text{Key.compare}(k, k') = \text{EQUAL}
(c) \text{lookup } k \text{ (insert } (k', v') \text{ } D) = \text{lookup } k \text{ } D
  \quad \text{if } \text{Key.compare}(k, k') \neq \text{EQUAL}
ordered types

structure Integers : ORDERED =
struct
  type t = int
  val compare = Int.compare
end

structure Strings : ORDERED =
struct
  type t = string
  val compare = String.compare
end

Int.compare and String.compare are comparison functions
functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

a dictionary functor

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......

(a dictionary functor

functor BSTDict (Key : ORDERED) : DICT =

struct
  structure Key = Key
  datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
  type 'a dict = 'a tree

val empty = Leaf

fun lookup k Leaf = NONE
  | lookup k (Node (L, (k', v'), R)) =
      case Key.compare (k, k') of
        EQUAL => SOME v'
        | LESS  => lookup k L
        | GREATER => lookup k R

......
fun insert (k, v) Leaf = Node(Leaf, (k, v), Leaf)
|    insert (k, v) (Node(L, (k', v'), R)) =
|        case Key.compare(k, k') of
|            EQUAL => Node(L, (k, v), R)
|            LESS => Node(insert (k, v) L, (k', v'), R)
|            GREATER => Node(L, (k', v'), insert (k, v) R)

fun trav Leaf = [ ]
|    trav (Node(L, (k,v), R)) = (trav L) @ (k,v) :: (trav R)
using a functor

• We defined a functor BSTDict for building implementations of DICT

• This encapsulates a common construction (binary search trees) in an abstract and general manner

• Easy to use, just apply the functor and re-use the code

- structure StringDict = BSTDict (Strings);
- structure IntDict = BSTDict (Integers);
invariant

“T is a Key.compare binary search tree”

Characterized inductively by:

- **Leaf** is a binary search tree
- **Node(L, (k,v), R)** is a binary search tree iff
  - every key in L is **LESS** than k
  - every key in R is **GREATER** than k
  - L and R are binary search trees
properties

• Let \( K : \text{ORDERED} \) be an ordered type, so \( K\.\text{compare} \) is a \textit{comparison function} for type \( K\.t \)

• Let \( S \) be the structure given by \( \text{BSTDict}(K) \)

• Then \( S \) implements \textit{DICT}

For all types \textit{entry}

 every value of type \textit{entry} \( S\.\text{dict} \)
definable from \( S\.\text{empty} \) and \( S\.\text{insert} \)
is a \( K\.\text{compare} \) binary search tree
why?

• S.empty is a binary search tree

• If T is a binary search tree, \( k : K.t \) and \( v : \text{entry} \), then \( S.\text{insert} \ (k,v) \ T \) is a binary search tree

• S.empty and S.insert are the only visible operations that produce dictionaries
  
  • Users can only build dictionaries with S.empty and S.insert

• So every dictionary that users can build is guaranteed to be a binary search tree
importance?

• *local* reasoning guarantees a *global* property

• correctness proof only needs to deal with the code *inside* the structure

• Users have limited access, so they *cannot* “*break the abstraction barrier*”

• the *invariant* always holds

*Every dictionary built from empty and insert is a binary search tree*
correctness

empty, insert, lookup
on binary search trees
satisfy “the usual equations”, such as:

(a) lookup k empty = NONE

(b) lookup k (insert (k’, v’) T) = SOME v’
   if Key.compare(k, k’) = EQUAL

(c) lookup k (insert (k’, v’) T) = lookup k T
   if Key.compare(k, k’) <> EQUAL

Exercise… prove this!
\( T_1, \ldots, T_4 : \text{int IntDict.dict} \)

\[ T_1 = \text{insert } (1,1) \text{ empty} \]

\[ T_2 = \text{insert } (2,4) T_1 \]

\[ T_3 = \text{insert } (3,9) T_2 \]

\[ T_4 = \text{insert } (4,16) T_3 \]
Standard ML of New Jersey v110.76 […] -

**structure** IntDict = BSTDict(Integers);
**open** IntDict;
**fun** build [ ] = empty
  | build (x::L) = insert (x, x*x) (build L) ;
**val** D = build [1,2,3,4,5];

ML says

val D = Node (Node (…,(5,25),Leaf)) : int dict
build \([1,2,3,4,5]\)

\[= \text{insert} \ (1,1) \ (\text{insert} \ (2,4) \ (\ldots (\text{insert} \ (5,25) \ \text{empty}) \ldots))\]

evaluates to

\[\text{Node} \ (\text{Node} \ (\ldots, (5,25), \text{Leaf})) : \text{int \ dict}\]

a binary search tree, but poorly balanced!
efficiency?

• The work to evaluate \text{lookup } k \ T on a binary search tree \ T is \(O(\text{size } T)\), in the worst case

\begin{itemize}
  \item (5, 25)
  \item (4, 16)
  \item (3, 9)
  \item (2, 4)
  \item (1, 1)
\end{itemize}

\text{lookup } 1 \ T

\text{lookup } 0 \ T
building better trees

• Next we’ll implement red-black trees

• Binary search trees with colored nodes and good enough (but not perfect) balance
  • The longest path length from root to leaf is no more than twice the shortest path

Work to evaluate lookup \( k_T \) on a red-black tree \( T \) will be \( O(\log (\text{size } T)) \)
what’s good?

not good

brown paths are longest, blue paths are shortest

5:1

4:2

3:2

good enough
good enough
why better?

• In a tree with $n$ nodes and a path ratio bounded by $2$, the cost of a lookup is $O(\log n)$.

• For ordinary binary search trees, the cost of a lookup is $O(n)$.

*decent balance implies fast lookup* (and insert)
red-black trees

datatype color = Red | Black

datatype 'a tree = Leaf
  | Node of 'a tree * (color * (Key.t * 'a)) * 'a tree

type 'a dict = 'a tree

int keys, string entries
red-black invariant
Roses are red,
Violets are blue,
Nodes can be red or black,
Leaves will be black.

Red nodes have black children,
The black-height’s the same.
However you say it
This poem is lame.
red-black invariant

Sorted: the tree is a Key.compare binary search tree

Well-red: no red node has a red child

Well-black: every path from root to a leaf has the same number of black nodes
  • this is the black height of the tree

bst
well-red
black height is 4

only keys and colors shown
red-black trees

• A red-black tree is a tree value that satisfies the invariant:
  - is a binary search tree
  - is well-red
  - is well-black

We will implement DICT using red-black trees
empty

val empty = Leaf

(This is a red-black tree!)

- ✅ bst
- ✅ well-red
- ✅ black height is 1
lookup

lookup : Key.t -> 'a dict -> 'a option

fun lookup k Leaf = NONE
   | lookup k (Node (L, (_, (k', v)), R)) =
      case Key.compare (k, k') of
         EQUAL       => SOME v
         LESS          => lookup k L
         GREATER  => lookup k R

(colors are ignored by lookup)
insertion

Two kinds: **updates** and **true insertions**

<table>
<thead>
<tr>
<th>insert ((k,v)) (D) is called an <strong>update</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>when (k) is EQUAL to a key already in (D)</td>
</tr>
<tr>
<td>• ((k,v)) will replace an existing entry</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>insert ((k,v)) (D) is called a <strong>true insertion</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>when (k) is not EQUAL to any key in (D)</td>
</tr>
<tr>
<td>• ((k,v)) will go in at a Leaf</td>
</tr>
</tbody>
</table>

**What should we do about colors?**
updates at root

- Updates don’t change the tree structure, so just \textit{copy} the color…

When \texttt{Key.compare}(k, k') = \texttt{EQUAL}
we’ll define

\[
\text{insert } (k, v) \ (\text{Node}(L, (c, (k', v'))), R)) = \text{Node}(L, (c, (k, v)), R)
\]
updates inside

When Key.compare\((k, k')\) = LESS
we need to put \((k, v)\) in *left* subtree, so

\[
\text{insert } (k, v) \ (\text{Node}(L, (c, (k', v'))), R))
= \text{Node}(\text{insert } (k, v) \ L, (c, (k', v'))), R)
\]

When Key.compare\((k, k')\) = GREATER
we need to put \((k, v)\) in *right* subtree, so

\[
\text{insert } (k, v) \ (\text{Node}(L, (c, (k', v'))), R))
= \text{Node}(L, (c, (k', v'))), \text{insert } (k, v) \ R)
\]

These operations will preserve the invariant!
true inserts at a Leaf

Let’s define

\[ \text{insert} \ (k, v) \ \text{Leaf} = \text{Node}(\text{Leaf}, (\textbf{Red}, (k, v)), \text{Leaf}) \]

The new entry replaces a leaf node, which was colored Black.

Choosing Black here would mess up black height if this leaf is in a larger tree.

- bst
- well-red
- black height is 1
true inserts

at a Node

When Key.compare(\(k, k'\)) = LESS
we need to put \((k, v)\) in left subtree, so

\[
\text{insert} \ (k, v) \ (\text{Node}(L, (c, (k', v'))), R))
= \text{Node}(\text{insert} \ (k, v) \ L, (c, (k', v'))), R)
\]

When Key.compare(\(k, k'\)) = GREATER
we need to put \((k, v)\) in right subtree, so

\[
\text{insert} \ (k, v) \ (\text{Node}(L, (c, (k', v'))), R))
= \text{Node}(L, (c, (k', v'))), \text{insert} \ (k, v) \ R)
\]

But will this maintain the invariants?
true insertion

example

insert (1,"1")

(3, "3")

(4, "4")

(5, "5")

a red-black tree

bst

well-red

black height is 2
= insert (1,"1") ( ( (3, "3") (4, "4") ) (5, "5") )
insert (1,"1") Leaf
not a red-black tree!

bst

well-red

black height is 2
restoring invariant

• The left-child is *almost* a red-black tree
  • keys are sorted,
  • paths have same black height
  • left-child *root* is red and has a red child

• That’s the only *invariant violation*

• We can *re-balance*, by *rotating* and *re-coloring*, to obtain a red-black tree...
balancing

rotate & re-color

(4,"4")

(3,"3")

(5,"5")

(1,"1")

(3,"3")

(4,"4")

(5,"5")

bst
well-black, bh = 2
almost well-red

bst
well-black, bh = 2
well-red

same lookup and traversal properties
more generally

For a **new insert** into a non-empty tree there are 4 awkward cases like this example:

1. **left** child has a **red root** and **red left** child
2. **right** child has a **red root** and **red left** child
3. **left** child has a **red root** and **red right** child
4. **right** child has a **red root** and **red right** child
balancing

rotate & re-color

and so on
(all four cases)
using patterns

Node(Node(Node(a, (Red, x), b), (Red, y), c), (Black, z), d)

(similarly for all four cases)
fun balance (Node(Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d))
    = Node(Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d)) ①
    | balance (Node(a,(Black,x), Node(Node(b,(Red,y),c), (Red, z), d))))
      = Node(Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d)) ②
    | balance (Node(Node(a, (Red,x), Node(b,(Red,y),c)), (Black,z), d))
      = Node(Node(a,(Black,x),b), (Red, y), Node(c,(Black,z),d)) ③
    | balance (Node(a, (Black,x), Node(b, (Red,y), Node(c,(Red,z), d))))
      = Node(Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d)) ④
    | balance T = T

do nothing except in awkward cases
So it seems like we should define

```haskell
fun insert (k, v) Leaf = Node(Leaf, (Red, (k, v)), Leaf)
| insert (k, v) (Node(L, (c, (k', v'))), R)) =
  case Key.compare(k, k') of
    EQUAL => Node (L, (c, (k,v)), R)
    LESS  => balance(Node (insert (k,v) L, (c, (k',v'))), R))
    GREATER => balance(Node (L, (c, (k',v'))), insert (k,v) R))
```

Very similar to bst insert, but uses `balance` in strategic places!

**Specification**

**REQUIRES** \(D\) is a red-black tree

**ENSURES** \(\text{insert} \ (k,v) \ D = \text{a red-black tree}...\)
still not right!

insert (1,"1")

= balance

= balance

red violation at root
It's easy to turn a tree with just a red violation at its root into a red-black tree...

💡 **blacken** its root!

(* \(\text{blackenroot} : \text{'a dict} \rightarrow \text{'a dict} \)*)

```ocaml
fun blackenroot Leaf = Leaf
| blackenroot (Node(L, (_, (k,v)), R))     = Node(L, (Black, (k,v)), R)
```

**restoring invariant**
fun \text{ins} \ (k, v) \ \text{Leaf} \ = \ \text{Node}(\text{Leaf}, \ (\text{Red}, \ (k, v)), \ \text{Leaf}) \\
| \ \text{ins} \ (k, v) \ (\text{Node}(L, \ (c, \ (k', v'))), \ R)) \ = \\
\ \text{case} \ \text{Key.compare}(k, k') \ \text{of} \\
| \ \text{EQUAL} \ => \ \text{Node} \ (L, \ (c, \ (k,v)), \ R) \\
| \ \text{LESS} \ => \ \text{balance}(\text{Node} \ (\text{ins} \ (k,v) \ L, \ (c, \ (k',v'))), \ R)) \\
| \ \text{GREATER} \ => \ \text{balance}(\text{Node} \ (L, \ (c, \ (k',v'))), \ \text{ins} \ (k,v) \ R)) \\
\text{fun} \ \text{insert} \ (k, v) \ D = \ \text{blackenroot} \ (\text{ins} \ (k, v) \ D)

We renamed the \textit{recursive inserting} function \textit{ins}...

\textit{insert} calls \textit{ins}, which \textit{balances} after recursive calls, then finishes with \textit{blackenroot}
\[ U_1 = \text{insert } (1,1) \text{ empty} \]

\[ U_2 = \text{insert } (2,4) \ U_1 \]

\[ U_3 = \text{insert } (3,9) \ U_2 \]

\[ U_4 = \text{insert } (4,16) \ U_3 \]
specifications

**red-black** means
binary-search tree, **well-red, well-black**

**almost-red-black** means
binary-search tree, **almost-well-red, well-black**

**well-red**: no red node has a red child

**almost-well-red**: no red node has a red child, except possibly the root node

**well-black**: every path has the same black height
specifications

\text{ins : Key.t * 'a -> 'a tree -> 'a tree}

\text{REQUIRES \quad D is red-black}

\text{ENSURES \quad \text{ins (k, v) D is almost-red-black}}
\text{with same black height as D…}

\text{balance : 'a tree -> 'a tree}

\text{REQUIRES \quad D is Node(A, x, B), D is bst,}
\text{A is almost-red-black, B is red-black \quad (or vice versa)}
\text{A and B have same black height}

\text{ENSURES \quad balance D is almost-red-black}
\text{with same black height as D…}
specifications

blackenroot : 'a tree -> 'a tree
REQUIRES \( D \) is almost-red-black
ENSURES blackenroot \( D \) is red-black

insert : Key.t * 'a -> 'a tree -> 'a tree
REQUIRES \( D \) is red-black
ENSURES insert \((k, v)\) \( D \) is red-black
balance spec

bst
well-black
left child almost well-red
right child well-red

bh(a) = bh(b) = bh(c) = bh(d)
• Traversal ignores color

```plaintext
fun trav Leaf = [ ]
  | trav (Node(L, (_, (k,v)), R))
    = (trav L) @ (k,v) :: (trav R)
```
The specs for the RBTDict code belong *inside the structure body*, not the signature!

Must include the *invariant*

Would be OK to omit type information that appears in the signature

Give types and specs for helper functions that are private to the implementation, as they do not appear in the signature
functor RBTDict(Key : ORDERED) : DICT =
struct
(* INVARIANT: every dictionary value built from empty and insert is red-black.
  Red-black means binary search tree & well-red & well-black.
  Well-red means no red node has a red child.
  Well-black means every path from root to a leaf
  has the same number of black nodes.
  This number is called the black height of the tree.
*)

structure Key : ORDERED = Key
datatype color = Red | Black
datatype 'a tree = Leaf | Node of 'a tree * (color * (Key.t * 'a)) * 'a tree
type 'a dict = 'a tree

(* ENSURES empty is red-black *)
val empty = Leaf
(* lookup : Key.t -> 'a dict -> 'a option
  REQUIRES D is red-black
  ENSURES lookup k D = SOME v  if there is an entry (k',v) in D
                               with Key.compare(k, k') = EQUAL
                               = NONE    otherwise
*)

fun lookup k Leaf = NONE
  | lookup k (Node (L, (_, (k', v')), R))  =
    case Key.compare (k, k') of
    | EQUAL          =>  SOME v'
    | LESS             =>  lookup k L
    | GREATER     =>  lookup k R
(* ALMOST-INVARIANT:
   A tree is almost-red-black if it is a bst, well-black,
   and no red node has a red child, except possibly the root *)

(* balance : 'a tree -> 'a tree
   REQUIRES D is red-black,
       or a non-empty well-black bst with
       an almost-red-black child and a red-black child
   ENSURES balance D is almost-red-black
   *)

fun balance ...

(* blackenroot : 'a tree -> 'a tree
   REQUIRES D is almost-red-black
   ENSURES blackenroot D is red-black
   *)

fun blackenroot ...
(* ins : Key.t * 'a -> 'a dict -> 'a tree
    REQUIRES D is red-black
    ENSURES ins (k, v) D is almost-red-black *)

fun ins (k, v) Leaf = Node(Leaf, (Red, (k, v)), Leaf)
| ins (k, v) (Node(L, (c, (k', v')), R)) =
  case Key.compare(k, k') of
    EQUAL     => Node (L, (c, (k,v)), R)
| LESS         => balance(Node (ins (k,v) L, (c, (k',v')), R))
| GREATER => balance(Node (L, (c, (k',v')), ins (k,v) R))

(* insert : Key.t * 'a -> 'a dict -> 'a dict
    REQUIRES D is red-black
    ENSURES insert (k,v) D is red-black *)

fun insert (k, v) D = blackenroot (ins (k, v) D)
Standard ML of New Jersey v110.76 [...] -

structure S = RBTDict(Integers);
open S;
fun build [ ] = empty
  | build (x::L) = insert (x, x*x) (build L) ;
val D = build [1,2,3,4,5];

ML says
val D =
    Node(Node(Node #,(#,#),Node #),
         (Black,(4,16)),
         Node (Leaf,(#,#),Leaf))
results

build [1,2,3,4,5] evaluates to

Node(Node(Node #,(#,#),Node #),
(Black,(4,16)),
Node (Leaf,(Black,(5,25)),Leaf))

a red-black tree

…decently balanced!
a correspondence

• Every value built from `empty` and `insert` is a **red-black tree**

• Ignoring colors, it’s also a **binary search tree**

```
Bst.insert (k_n,v_n) (...(Bst.insert (k_1,v_1) Bst.empty)...) and
Rbt.insert (k_n,v_n) (...(Rbt.insert (k_1,v_1) Rbt.empty)...) represent the same dictionary
but the red-black version has better work/span
```
• Let $T$ be a bst, $U$ a red-black tree

• Say $T \approx U$ iff their lookup functions are equal

• empty bst $\approx$ empty red-black tree

• If $T \approx U$ then $\text{insert} \ (k, v) \ T \approx \text{insert} \ (k, v) \ U$