1 Topics

• Modular programming.
• Functors can encapsulate a common construction.
• Information hiding, abstract types, and representation invariants.
• Local reasoning can ensure a global property.

2 Overview

We begin by presenting a signature for an abstract type of dictionaries, with operations for inserting items (tagged with a key), and looking up the item associated with a key. Keys belong to an ordered type. We can implement dictionaries as sorted lists of key-value pairs, or as binary search trees with key-value pairs at nodes. We use functors to encapsulate these implementations: the same code designs work uniformly across a wide range of applications.

The binary search tree implementation still suffers from poor worst-case behavior, so this motivates us to explore a more sophisticated tree-based implementation (red-black trees). We give careful specifications that help us

*Building on notes by Dan Licata.
when we develop the code. We discuss why the red-black tree implementation achieves better worst-case behavior.

The self-test questions ask you to reflect on what you learned about using signatures and structures, and on the consequences of using signatures transparently or opaquely.

## 3 Dictionaries

We begin with two signatures:

- **ORDERED**, an interface providing a type named \( t \) and a “comparison” function of type \( t \times t \rightarrow \text{order} \).

- **DICT**, an interface with a parameterized type \( 'a \text{ dict} \), a structure \( \text{Key} \) with signature **ORDERED**, a value \( \text{empty} \) and functions \( \text{insert} \) and \( \text{lookup} \). We also include a function \( \text{trav} \) that’s useful for debugging.

```ocaml
signature ORDERED =
  sig
    type \( t \)
    val compare : \( t \times t \rightarrow \text{order} \)
  end

signature DICT =
  sig
    structure \( \text{Key} \) : ORDERED
    type \( 'a \text{ dict} \)
    val empty : \( 'a \text{ dict} \)
    val insert : \( \text{Key}.t \times 'a \rightarrow 'a \text{ dict} \rightarrow 'a \text{ dict} \)
    val lookup : \( \text{Key}.t \rightarrow 'a \text{ dict} \rightarrow 'a \text{ option} \)
    val trav : \( 'a \text{ dict} \rightarrow (\text{Key}.t \times 'a) \text{ list} \)
  end
```

Note: it’s OK to have a signature containing a structure! That will mean that when we implement this signature, we’ll be building a structure containing another structure. Also note: we decided to curry the function types in the DICT interface. When we implement, we’ll need to be careful to curry in the right places, too!

It’s easy to build a structure that implements **ORDERED**. For example:
structure Integers : ORDERED = 
  struct
    type t = int
    val compare = Int.compare
  end;

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

The type Integers.t is int, and Strings.t is string.

4 Association lists

One way to implement the signature DICT is to represent a dictionary as an 
association list: a (sorted-by-keys) list of key-value pairs. Since the same 
idea works for any choice of an ordered type (for keys), it is natural to define 
a functor:

functor AssocDict(Key : ORDERED) : DICT = 
  struct
    structure Key = Key;
    type 'a dict = (Key.t * 'a) list
    val empty = [ ]
    fun insert (k, v) D = ...
    fun lookup k D = ...
    fun trav D = D
  end

In the functor body (which we have only sketched) we use key-sorted lists 
of key-value pairs to represent dictionaries. The empty dictionary is just the 
empty list; the insertion function must respect the key ordering – you can 
adapt the helper function we used earlier in defining insertion sort. Then 
lookup k can take advantage of the ordering in the obvious way – search 
the list for an entry with a key that’s Key.compare-EQUAL to k, and stop if 
you reach a key that’s Key.compare-GREATER. (You can fill in the missing 
code.)
We can use the functor easily to generate structures that implement the
dictionary signature:

structure IntDict : DICT = AssocDict(Integers);

structure StringDict : DICT = AssocDict(Strings);

Try doing this (after you filled in the missing details in the functor body).
Try building some example dictionaries and evaluating some expressions that
manipulate them. For example:

open IntDict;

val D1 = insert (1,’’foo’’) (insert (2, ‘’bar’’) empty;

val x = lookup 1 D1;
val y = lookup 2 D1;
val z = lookup 42 D1;

• What would happen if we ascribe the signature opaquely, e.g.

  structure IntDict :> DICT = AssocDict(Integers);

  structure StringDict :> DICT = AssocDict(Strings);

• Figure out the work for insert (k, v) D and lookup k D when D is
a (sorted) list of length n.

If you filled in the missing code details properly, you should be able to show
the following properties for the association lists implementation:

1. lookup k empty = NONE

2. lookup k (insert (k’, v’) D) = SOME v’
   if Key.compare(k, k’) = EQUAL

3. lookup k (insert (k’, v’) D) = lookup k D
   if Key.compare(k, k’) <> EQUAL.
5 Binary search trees

We will now define a structure that implements dictionaries as binary search
trees of key-value pairs. Again we do this as a functor, since the same
construction works uniformly, no matter what the ordered type of keys is.
(This time we include all the details.)

functor BSTDict(Key : ORDERED) : DICT =
  struct
    structure Key : ORDERED = Key;
    datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree;
    type 'a dict = 'a tree;

    (* empty : 'a dict *)
    val empty = Leaf;

    (* lookup : Key.t -> 'a dict -> 'a option *)
    fun lookup k Leaf = NONE
    | lookup k (Node (D1, (k', v'), D2)) =
      case Key.compare (k,k') of
      | EQUAL => SOME v'
      | LESS => lookup k D1
      | GREATER => lookup k D2

    (* insert : Key.t * 'a -> 'a dict -> 'a dict *)
    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)
  end

Note that the signature ascription (: DICT) will prevent clients from using
Leaf or Node explicitly.
Using this functor

Here is some ML code that uses this functor, opens it so we can use the dictionary operations at the top level, and defines a function for creating a dictionary from an association list.

structure S = BSTDict(Integers);
open S;

fun build [ ] = empty
| build (x::L) = insert (x, Int.toString x) (build L)

However, this implementation gives no guarantees about balance! For example:

- val T = build [1,2,3,4,5,6];
val T = Node (Node (Node #,(#,#),Leaf),(6,"6"),Leaf) : string tree

(As we ascribed the signature transparently, and we opened S, ML will report the type of T here as string tree, even though the type name tree is not mentioned in DICT. We could just as well have gotten T : string S.dict.)

In pictorial form, this tree T looks like

```
               (1,1)
               /  \
(2,2)         (3,3)
           /   \    /   \\
(4,4) (5,5) (6,6)
```

Not very balanced! Lookup in an unbalanced tree is going to have worst-case runtime approximately the length of the longest path. (Same as with
Clearly we’d prefer to deal with balanced trees. But it’s going to be expensive to revise the insertion function so that it preserves balance. (Try it!) Next we’ll explore a way to implement DICT that yields better asymptotic runtime, by guaranteeing a form of balance.

Before we move on, note that the binary search tree implementation of dictionaries also validates the same properties as the association lists implementation did:

1. lookup k empty = NONE
2. lookup k (insert (k’, v’) D) = SOME v’
   if Key.compare(k, k’) = EQUAL
3. lookup k (insert (k’, v’) D) = lookup k D
   if Key.compare(k, k’) <> EQUAL.

And it’s also easy to see that for all binary search trees D, the value of trav D is a list of key-value pairs that’s Key.compare-sorted with respect to keys. (This was trivially true for association lists, but needs proof here.)

6 Red-black trees

For a given list of key-value pairs, there are many different binary search trees containing the data in this list. Some are poorly balanced, and some are better balanced. The problem with unbalanced trees is that in the extreme case (when all the data is strung out on a single path) lookups and inserts take time linear in the number of items in the dictionary. With a decently balanced tree we’d expect logarithmic time lookups and inserts. It would be hard to implement insertion in such a way that we always obtain a perfectly balanced tree (allowing for the fact that the number of items in the tree may be odd or even, and may not be close to a power of 2). In fact it would probably be too expensive to strive for optimal balance, and we won’t even try. The problem becomes even more difficult if we extend our implementation to include a deletion operation as well.

Instead, we will examine a well known implementation of red-black trees, which guarantees a weaker but decent enough and useful kind of balance: in a red-black tree the ratio of the length of the longest path to the length of the shortest path is at most 2-to-1. So grossly imbalanced trees like the one
we saw earlier time will never arise! With this implementation, even though not perfectly balanced, lookups and inserts do indeed take logarithmic time.

The main ideas are as follows. We work with binary trees whose data are key-value pairs tagged with a “color”. There are two possible colors (red and black!) and some rules to be obeyed when building trees.

A binary tree of color-key-value data is a red-black tree iff each node is colored red or black, leaf nodes are black, and the tree is:

1. **Sorted**: The keys are sorted with respect to key-comparison, just as in the usual binary search tree invariant: keys in the left subtree are **LESS** than the key at the root, etc;

2. **Well-black**: Each path from the tree’s root to a leaf contains the same number of black nodes (we call this number the tree’s black **height**);

3. **Well-red**: No red node has a red child.

If all nodes are black, the well-black property would imply that the tree is perfectly balanced. The red-black properties imply that the longest path from root to leaf is no more than double the length of the shortest path from root to leaf; so a red-black tree with \( n \) nodes has height \( \Theta(n) \), and can be searched in logarithmic time. By making dictionaries this way, and only providing operations that preserve the red-black invariant, we guarantee that users always obtain balanced trees!

Moreover, the red-black invariant is easy to maintain, needing just a slight modification (carefully chosen) to our insertion operation, as we will soon see.

The key ideas that we believe this development shows are:

(a) **Representation invariants help us clarify design intentions, specify and prove correctness, and achieve efficiency.**

(b) **An abstract type supports local reasoning about correctness and efficiency, without needing to worry about client behavior.**

The red-black properties (bst, well-red and well-black) are the ingredients in a “representation invariant” that we build into the implementation. Every dictionary constructible by using the visible operations (empty and insert) is guaranteed to satisfy this property. And we can reason “locally” (referring solely to the data and functions inside the structure or the functor body). This is hugely important! And justified, because there’s no way any user can
break the representation invariant, because of the guarantee above and the scope rules governing the visibility of the types and operations defined in our module.

To build a better dictionary implementation we'll define a functor

\[ \text{RBTDict} \]

that can be applied to a structure \[ \text{Key} : \text{ORDERED} \] to build a structure with signature \[ \text{DICT} \]. So we'll need to include in the functor body a definition for the type \[ \text{'a dict} \], definitions for \[ \text{empty:'a dict}, \text{insert}, \text{and so on.} \]

Inside the functor body we'll define a datatype \[ \text{color} \] for colors:

\[
\begin{align*}
\text{datatype color} & = \text{Red} | \text{Black} \\
\text{and we'll implement \text{'a dict} as binary trees with color-key-data entries of type \text{color} \ast (\text{Key.t} \ast \text{'a}) at the nodes:} \\
\text{datatype \text{'a tree} = Leaf | Node of \text{'a tree} \ast (\text{color} \ast (\text{Key.t} \ast \text{'a})) \ast \text{'a tree} \\
\text{type \text{'a dict} = \text{'a tree}} \\
\end{align*}
\]

(Since \text{Leaf} and \text{Node} are not in the signature \text{DICT}, users won't be able to use them. But we'll be OK using them inside our structure definition.)

We can implement the empty dictionary as before:

\[ \text{val empty = Leaf} \]

And the lookup function doesn't need to pay attention to colors, so we can adapt the old function definition in the obvious way:

\[
\begin{align*}
\text{(* lookup : Key.t -> \text{'a dict} -> \text{'a option *)} \\
\text{fun lookup k Leaf = NONE} \\
\text{| lookup k (Node (D1, (_, (k',v)), D2)) =} \\
\text{case Key.compare (k, k') of} \\
\text{EQUAL} & => \text{SOME v} \\
\text{| LESS} & => \text{lookup k D1} \\
\text{| GREATER} & => \text{lookup k D2}
\end{align*}
\]

We used an underscore pattern to emphasize the fact that colors at nodes are ignored by the lookup function.

As you should expect, in designing the insertion operation we need to pay close attention to color. We need to design
insert : Key.t * 'a -> 'a dict -> 'a dict

to ensure that, whenever D is a dictionary value satisfying the red-black-tree
properties, then for all key-value pairs (k,v), insert (k, v) D returns a
dictionary value that also satisfies the red-black-tree properties. How should
we achieve this?

There are two kinds of insertion situations, which differ in how they affect
tree structure: “new” inserts, of the form insert D (k, v) in which k is
not EQUAL to any key in D; and “updates”, in which k is EQUAL to some key
already in D. New inserts actually take effect at leaf nodes, and updates don’t
alter the tree structure at all.

Let’s start by echoing the original insertion function for binary search
trees, but let’s rename the function ins in the expectation that we’re going
to have to tweak it later to obtain the final insert function for red-black
trees:

fun ins (k, v) Leaf = Node(Leaf, (k, v), Leaf)
| ins (k, v) (Node(l, (k', v'), r)) =
  case Key.compare(k, k') of
    EQUAL => Node(l, (k,v), r)
  | LESS => Node(ins (k,v) l, (k',v'), r)
  | GREATER => Node(l, (k',v'), ins (k,v) r)

We must adjust to take account of colors, and the first issue is: what color
to use on the right-hand-side of the first clause? What color should we
use for the single-node tree built by inserting (k,v) into the empty tree?
Remembering that “new” inserts happen at leaf nodes, and that’s what we’re
defining here, it would be a bad idea to choose Black: this would mess up
the black height on the side of the tree where the insert is going. So, let’s use
Red, since that would obviously give us a tree with the required properties
(1), (2) and (3). So our new first clause is going to be:

ins (k, v) Leaf = Node(Leaf, (Red, (k, v)), Leaf)

The second clause of the function also needs to be adjusted, because the tree
argument on the left-hand-side has a color field and we need to specify a
color on the right-hand-side. Here’s the obvious first attempt, which is also
obviously not going to give us good balance but at least has the virtue of
being well typed and maintaining sortedness:
ins (k, v) (Node(l, (c, (k', v')), r)) =
case Key.compare(k, k') of
   EQUAL     => Node(l, (c, (k,v)), r)
| LESS      => Node(ins (k,v) l, (c, (k',v')), r)
| GREATER   => Node(l, (c, (k',v')), ins (k,v) r)

This just copies the same color over to the root on the right-hand-side!
It’s not hard to see that (with the function defined by these two clauses) if
we start with a red-black tree and k is EQUAL to k’ (i.e. an update) the result
will indeed be a red-black-tree, because we return another tree with the same
shape and colors. So “updates” work fine, and preserve the representation
invariant.

But this simple-minded color-propagation idea doesn’t work for “new”
insertions, because we will be putting a new Red-colored node at a leaf of
the tree; even if the original tree was red-black, there’s no reason why adding
this node should also produce a red-black-tree. (Because we only insert a
red node, the black-height of the tree stays the same. We also preserve
sortedness. But the well-red part of the invariant may not hold any more.)
We can see this with an example.

If we insert (1, "1") into the red-black-tree

(Black, (4,"4"))
  /     \
(Red, (3, "3")) (Red, (5, "5"))
  / \      / \
 . . . .

using the function as given above, we’ll get

(Black, (4, "4"))
  /     \
(Red, (3, "3")) (Red, (5, "5"))
  /     \      /     \
(Red, (1, "1")) . . .
  /     \. . .

which has a red node with a red parent, so isn’t a red-black-tree.

Don’t panic! It’s OK if we find ourselves violating the representation
invariant somewhere inside the implementation, as long as we restore the
invariant before returning. This issue introduces the important idea of a critical section of code: inside the implementation of a module, you can break the representation invariant in a piece of code, provided you repair the invariant before returning any values to clients. (The code fragment in which it is OK to relax the invariant temporarily is known as a critical section.) From the client’s viewpoint, insert takes a RBT and produces a RBT. But internally, in the process of doing an insert, it’s OK to work temporarily with trees that do not satisfy the invariants.¹

In this example, there’s an obvious easy way to fix it up: just rotate the tree at the root and adjust the colors, to get:

\[
\begin{align*}
\text{(Red, (3, "3")}) \\
/ \\
(\text{Black, (1, "1")}) (\text{Black, (4, "4")}) \\
/ \ \\
. . . . (\text{Red, (5, "5")}) \\
/ \\
. . .
\end{align*}
\]

This trick for restoring some balance by rotating and re-coloring in this example turns out to be a special case of a more general fix that we’ll explain shortly.

Similarly, if we naively insert \((1, "1")\) into a RBT with a single node, a red root, such as

\[
\begin{align*}
\text{(Red, (3, "3")}) \\
/ \\
. .
\end{align*}
\]

we get an almost-well-red tree as a result,

\[
\begin{align*}
\text{(Red, (3, "3")}) \\
/ \ \\
(\text{Red, (1, "1")}) . \\
/ \ \\
. .
\end{align*}
\]

¹Analogy: the representation invariant is “your room is clean”. The clients are your parents. During the semester, your room can be dirty, as long as you clean it before parents’ weekend and the end of the year.
and again there is an easy fix: blacken the root color, to get

\[
\begin{array}{c}
\text{(Black, (3, "3"))} \\
/ \\
\text{(Red, (1, "1"))} \\
/ \ \\
. .
\end{array}
\]

(This is obviously a red-black-tree!)

We need to allow invariant violations of the following kind: we will allow temporary use of trees that are *almost-red-black*. An ARBT is like a RBT (well-black, and sorted), except that instead of being well-red, it is

\[
\text{(almost-well-red): no red node has a red child, except possibly the root node}
\]

As we have seen, our \texttt{ins} function as defined above may produce a tree whose left- or right-child is almost red-black; and there are four possible situations that cover all ways for this to happen: left- or right child with the defect; and in each case, either the left or right child of the defective subtree could be the “red child of a red node”. We can \textit{rebalance} in such cases, to produce a tree that is well-red and without messing up the black-height or sortedness of the tree. ² We can make these ideas precise by defining an ML function

\[
\texttt{balance : 'a tree -> 'a tree}
\]

that we will let \texttt{ins} use strategically. For each of these 4 ways for the shape of a tree

\[
\text{Node (T1, (c, (k,v)), T2)}
\]

to cause trouble, we can define a \textit{pattern} that matches exactly the trees with the bad color/shape combination, and in each case there is a simple “rotation” that can be done to solve the problem: \textit{rotate} at the root, making the root red with two black children, while preserving the sortedness and well-blackness of the tree, \textit{and} ensuring that there are no well-red violations \textit{except possibly at the root}.

\footnote{This balancing scheme is due to Chris Okasaki (\textit{Red-Black Trees in a Functional Setting}, Journal of Functional Programming, 1999).}
Consider the case of a tree with a black root, where the left child is an ARBT, the right is an RBT, the left-left child is a red child of a red node, and the tree is sorted. Let’s draw this pictorially as

```
(Black, z)
 /   \
(Red, y)  d
 /   \
(Red, x)  c
 /   \
 a   b
```

To balance this picture we rotate at the root and re-color, to get

```
(Red, y)
 /   \
(Black, x) (Black, z)
 /   \   /   \
 a   b   c   d
```

Why does this work? We need to show that the new tree has the three desired properties:

- **Sortedness:** because the original tree was assumed to be sorted, the keys in \( c \) are greater than or equal to the key in \( y \), but less than or equal to the key in \( z \), so the new tree with \( c \) as can the left child of \( z \) will also be sorted.

- **Well-red:** by assumption, \( a, b, c, d \) are red-black trees; a black node can have any red-black trees as children, so the trees rooted at \( x \) and \( z \) are well-red. Because \( x \) and \( z \) are colored black, the overall tree is a red-black tree. (Note that this works even if the root of \( c \) is also red, which is possible — in an almost red-black tree, both children of a red root might be red.)

- **Black-height:** By assumption, the original tree has a black-height, which must be positive, say \( h + 1 \), because the root is black; hence each of \( a, b, c, d \) has a black-height of \( h \). Thus, the result tree also has a black-height of \( h + 1 \) because each child of the red root is black.
You might like to consider if there are any other plausible choices of coloring that could have been made here.  

The correctness proofs for the other cases of rebalancing are analogous, by symmetry.

Notice that the almost red-back trees whose shape resembles the picture shown above will match the following ML pattern:

\[
\text{Node (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d)}
\]

where we use variable patterns \(a, b, c, d\) to match against subtrees, and variable patterns \(x, y, z\) to match against color-key-value data. And — with these variables bound to pieces of a tree — we can obtain the rotated tree simply by evaluating

\[
\text{Node (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d))}
\]

Here is the ML code for balancing, based on this idea.

There are four non-trivial cases, as above; in all other cases we leave the components alone.

\[
\begin{align*}
\text{fun balance (Node (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d))} &= \text{Node (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d))} \\
| \text{balance (Node (Node(a, (Red,x), Node(b,(Red,y),c)), (Black,z), d))} &= \text{Node (Node(a,(Red,x), Node(b,(Red,y),c)), (Black,z), d))} \\
| \text{balance (Node (a,(Black,x), Node(Node(b,(Red,y),c), (Red, z), d)))} &= \text{Node (Node(a,(Black,x), Node(Node(b,(Red,y),c), (Red, z), d)))} \\
| \text{balance T} &= \text{T}
\end{align*}
\]

So now we’re trying to work with the following insertion function:

---

3) Alternative colorings: We cannot make \(z\) red, because the root of \(d\) might be red. We could make \(x\) red and the root \(y\) black, but this would not satisfy the black-height invariant: if the black-height of the input is \(h + 1\), then the number of black nodes on the path to a leaf in \(b\) would be \(h + 1\), whereas the number of black nodes on a path to a leaf in \(d\) would be \(h + 2\). We could make each of \(x\) and \(y\) and \(z\) black; in this case, the result would have a black-height that is one more than the black-height of the input. However, the call site of rebalancing requires that it preserves the black-height, rather than incrementing it.

15
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))

But there’s one last problem. This doesn’t necessarily produce a red-black tree. It almost works; it definitely produces an almost-red-black tree. So all we need to do to finish the job is to make the color at the root be Black. Let’s introduce a helper function to do this:

(* blackenroot : 'a dict -> 'a dict *)
fun blackenroot Leaf = Leaf  
| blackenroot (Node(D1, (_, (k,v)), D2)) = Node(D1, (Black, (k,v)), D2)

Whenever D is an almost red-black tree, blackenroot D is a red-black-tree, because: making the root black obviously maintains sortedness, removes the only possible well-redness violation, and maintains the fact that the tree has a black-height, even though it potentially increases the black-height by one.

So our finished product, the insert function, will use ins followed by blackenroot:

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(balance(l, (c, (k',v')), ins (k,v) r))

(* insert : Key.t * 'a -> 'a dict ->'a dict *)
fun insert (k, v) D = blackenroot (ins (k, v) D)

Thus, whenever D is a red-black tree, so is the result of insert (k, v) D.

Comment
Although the representation invariants in this example are a little involved, the code is nice and clean— and the rebalancing function makes good use of pattern-matching.
To complete the functor definition we modify the traversal function in
the obvious way, omitting colors. (Why is this the right thing to do?) Here
is the function definition:

```plaintext
def fun trav Leaf = [] | trav (Node(l, (_, (k,v)), r)) = (trav l) @ (k,v) :: (trav r)
```

Draw pictures of the red-black trees that represent the results of building:

- val U0 = empty
- val U1 = insert U0 (1, "1")
- val U2 = insert U1 (2, "2")
- val U3 = insert U2 (3, "3")

7 Notes

In the RBTDict implementation we used helper functions like balance and
ins that are not listed in the signature DICT. That prevents users from using
these helpers. We also used a datatype in the implementation and kept the
constructors Leaf and Node private, again so that users cannot build any
dictionary values except by starting from empty and doing insert opera-
tions. This example is an excellent illustration of the benefits of information
hiding and data abstraction. We’ve designed the RBTDict implementation
to guarantee that all user-built dictionaries will actually be represented as
binary search trees with sufficiently well balanced tree structure to ensure
that lookups and insertions are fast!

8 Comparing two implementations

Suppose we build a red-black tree implementation of dictionaries, and a
binary-search tree implementation of dictionaries. Each is a structure with
the same signature, Dict. So each one allows users to build an empty diction-
ary and perform a sequence of insertions, to search the dictionary for an
entry with a key “equal” to a given key, and also to extract the sorted list of
key-value entries at any stage.
How, if at all, could a user distinguish between the two implementations? Well, we’ve shown that the red-back tree version is asymptotically faster. But in practice, unless we’re able to run a large-scale comparison on a huge sized dictionary, with instrumentation tools for assessing runtime, it’s probably not going to be easy to notice much difference that way. And if we just ask for “visible” results, i.e. we check the traversal lists produced using both versions after exactly the same sequences of insertions, we’ll never be able to observe a difference!

The reason is that we can prove a theorem that makes precise the sense win which the two implementations are observably equivalent. Since the only visible operations are empty, insert, lookup, and trav, what we need are some lemmas that link the observable behavior of these functions in the two implementations. Since the actual types used inside these implementations are different (one with colors, one without) we need to be careful to avoid confusion. Luckily we can use qualified names to keep things straight!

In what follows we will limit attention to dictionaries whose entries are strings, with integer keys. Of course the ideas and results hold much more generally.

Suppose Bst:DICT and Rbt:DICT are a binary-search tree dictionary implementation and a red-black tree dictionary implementation, defined by

\[
\text{structure Bst : DICT = BSTDict(Integers);} \\
\text{structure Rbt : DICT = RBTDict(Integers);} 
\]

The only ways to produce “observable” results from a dictionary value are to apply lookup, or trav. So we’ll say that a pair of values

\[(L, R) : \text{string Bst.dict} \times \text{string Rbt.dict}\]

is observationally equivalent iff

(i) For all values k:int, Bst.lookup L k = Rbt.lookup R k.
(ii) Bst.trav L = Rbt.trav R.

So observational equivalence means giving the same lookup results and the same traversal list.

The following results can be proven:

- Bst.empty and Rbt.empty are observationally equivalent.
• Whenever \( L \) and \( R \) are observationally equivalent, so are \( \text{Bst.insert}(k, v) \) \( L \) and \( \text{Rbt.insert}(k, v) \) \( R \).

And as a corollary, because of the fact that \textit{only} the empty value and the insert function are offered to clients of this abstract type, there’s no way for clients to tell the two implementations apart. (If you think that users could guess which is which based on the names we gave to the structures, that’s naïve. We could have been maliciously used the wrong names, or invented completely misleading names.)
9 Self-test

1. For the binary-search-tree implementation of dictionaries, is it true that for all suitably typed dictionary values $D$ the following equation holds, when $\text{Key.compare}(k, k') <> \text{EQUAL}$?

\[
\text{insert } (k, v) \text{ (insert } (k', v') D) \\
= \text{insert } (k', v') \text{ (insert } (k, v) D)
\]

In other words, does the order of the two insertions with different keys make no difference to the binary tree structure?

HINT: Don’t waste time trying to prove this is true; look for a simple counterexample instead!

What can you say about the value of $\text{insert } (k, v) \text{ (insert } (k', v') D)$ when $\text{Key.compare}(k, k') = \text{EQUAL}$?

2. For the binary-search tree implementation of dictionaries, prove the following properties (for all well-typed instances, assuming $\text{Key}$ is an ordered type):

(a) $\text{lookup } k \text{ empty } = \text{NONE}$
(b) $\text{lookup } k \text{ (insert } (k', v') D) = \text{SOME } v' \text{ if } \text{Key.compare}(k, k') = \text{EQUAL}$
(c) $\text{lookup } k \text{ (insert } (k', v') D) = \text{lookup } k D \text{ if } \text{Key.compare}(k, k') <> \text{EQUAL}$.

3. Show that for all red-black trees $D$ and keys $k$,

\[
\text{lookup } k \text{ (balance } D) = \text{lookup } k D \\text{ lookup } k \text{ (blackenroot } D) = \text{lookup } k D \\text{ trav } (\text{balance } D) = \text{trav } D \\text{ trav } (\text{blackenroot } D) = \text{trav } D
\]
4. Although a red-black tree isn’t guaranteed to be perfectly balanced, the lookup and insertion operations on a red-black tree are asymptotically “better” than the same operations on arbitrary binary search trees. Recall that the worst-case time to do an insertion on a binary search tree with \( n \) nodes is \( O(n) \), since it can take time proportional to \( n \) to insert into a badly unbalanced binary tree, which has depth \( O(n) \).

Prove the following properties.

- Every subtree of a red-black tree is also a red-black tree.
- For a red-black tree with \( n \) nodes and black-height \( b \), \( 2^b \leq n + 1 \).
- A red-black tree with \( n \) nodes has height \( O(\log n) \).

Explain why it follows from these results that the worst-case work to insert into a red-black tree with \( n \) nodes is \( O(\log n) \).

5. Use the RBTDict functor to build dictionaries:

(a) with string entries, integer keys, ordered by >
(b) with integer entries, string keys, ordered lexicographically

6. Examine the effect on users of ascribing the signature \texttt{DICT} opaquely instead of transparently. Do the same for the BSTDict functor.

7. We could have chosen a slightly different way to implement binary search trees (or, just as easily, red-black trees) along the following lines:

```haskell
functor BSTDict(Key : ORDERED) : DICT =
  struct
    structure Key = Key
    datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
    type 'a dict = (Key.t * 'a) tree
    ...
  end
```

Compare and contrast with the original way. Do you see any significant benefits?
8. Instead of defining a datatype for binary trees inside the structures, as we did for BSTDict and RBTDict, we could have done the following. We could have defined a binary tree datatype at the top level (globally) and then modified the functors:

```ml
datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

functor BSTDict(Key : ORDERED) : DICT =
  struct
    structure Key = Key
    type 'a dict = (Key.t * 'a) tree
    ...
  end
```

Why would this not be a good idea? Would it be any better, or different, if we used opaque ascription?

```ml
datatype 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

functor BSTDict(Key : ORDERED) :> DICT =
  struct
    structure Key = Key
    type 'a dict = (Key.t * 'a) tree
    ...
  end
```

9. When implementing an abstract type using a representation invariant it’s often a good idea to test your implementation to check that the invariant really does hold. (Of course it’s even better to prove it!) Inside your RBTDict functor body, implement some helper functions for testing to check that your code really does produce red-black trees. Declare an exception RBTError, to be used to signal a violation. Implement functions

```ml
bst : 'a dict -> bool
well_red : 'a dict -> bool
well_black : 'a dict -> bool
black_height : 'a dict -> int
```
red_black_tree : 'a dict -> bool
almost_red_black : 'a dict -> bool

with the obvious specifications. For example, well_red must be total,
and returns true if its argument is a well-red tree, false otherwise.
The spec for black_height is as follows:

REQUIRES true
ENSURES black_height D = n where n is the black height of D,
    if D is well-black
    black_height D = raise RBTError
    if D is not well-black

HINT: use black_height to help with well_black, and make use of
exception handling!