J. Functional Programming **11** (4): 425–432, July 2001. Printed in the United Kingdom
© 2001 Cambridge University Press

Red-black trees with types

STEFAN KAHRS
University of Kent at Canterbury, Canterbury, Kent, UK

Abstract

Chris Okasaki showed how to implement red-black trees in a functional programming language. Ralf Hinze incorporated even the invariants of such data structures into their types, using higher-order nested datatypes. We show how one can achieve something very similar without the usual performance penalty of such types, by combining the features of nested datatypes, phantom types and existential type variables.

1 Introduction

Red-black trees are a well-known way of implementing balanced 2-3-4 trees as binary trees. They were originally introduced (under a different name) in Bayer (1972) and are nowadays extensively discussed in the standard literature on algorithms (Cormen *et al.*, 1990; Sedgewick, 1988).

Red-black trees are *binary* search trees with an additional ‘colour’ field which is either *red* or *black*. In a proper red-black tree each red-coloured node is required to have black subtrees and is also regarded as an intermediate auxiliary node. Therefore, every black node has (possibly indirectly) either 2, 3 or 4 black-coloured subtrees, depending on whether it has 0, 1 or 2 red-coloured direct subtrees. This is the reason why red-black trees can be seen as implementation of 2-3-4 trees.

Red-black trees realise 3- and 4-nodes by connecting binary nodes. While this (at worst) doubles the height of the tree, compared to the associated 2-3-4 tree, it does not affect the number of comparisons a search has to make, and it simplifies the balancing process considerably.

Okasaki (1998, 1999) showed how this data structure can be implemented in a functional setting. An earlier attempt at implementing the rather similar 2-3 trees was made by Chris Reade (1992). Okasaki’s implementation is much more concise than the known imperative implementations and consequently much easier to understand. Figure 1 shows the definition of the type and Okasaki’s insertion[1] function.

It is worth iterating the basic invariants of red-black trees:

- every red node has two black children, with E being regarded black as well;

[1] This is the insertion operation when red-black trees are used to implement *sets*. For simplicity, we stick with this particular application.

```
data Color  =  R | B
data Tree a  =  E | T Color (Tree a) a (Tree a)

insert       :: Ord a => a → Tree a → Tree a
insert x s   = T B a y b
               where
               ins E                   =  T R E x E
               ins s@(T color a y b)   =
                      if x < y then balance color (ins a) y b
                      else if x > y then balance color a y (ins b)
                      else s
               T _ a y b               =  ins s

balance :: Color → Tree a → a → Tree a → Tree a
balance B (T R (T R a x b) y c) z d   =  T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d   =  T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))   =  T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)   =  T R (T B a x b) y (T B c z d)
balance c a x b                       =  T c a x b
```

Fig. 1. Okasaki's insertion algorithm.

- every path from the root to an empty tree passes through the same number of black nodes. (I will call this number the 'height' of the tree, as it is the height of the associated 2-3-4 tree.)

However, the algorithm maintains only the second invariant slavishly. The first is slightly weakened: red nodes *at the root of a tree* may have red children – we will call such trees 'infrared' (the other trees are 'proper'). The *balance* function assumes and promises the following:

- both tree arguments have the same height n;
- at least one tree argument is proper;
- if we balance with R then additionally neither argument is infrared;
- the result has height n if we balance with R and $n + 1$ otherwise;
- if we balance with B the result is proper.

Notice that Okasaki's type for red-black trees does not incorporate the invariants we demand. While the colour changes are easily enforceable, the balancing is a bit more delicate. Hinze (1999a, 1999b) showed how one can achieve even that. For red-black trees the resulting type definitions would be as in figure 2.

The type `RB` is a nested higher-order type: it is *nested*, because its first argument changes in its recursive occurrence (ruling out ML-like type systems (Kahrs, 1996)); it is higher-order, because that same argument has kind $* \to *$.

It is not particularly easy to write recursive functions that operate on such types. The change in the second type argument regularly requires a similarly changing argument for recursive functions operating on that type – and this argument is typically a function, or even a collection of functions. The function member has a functional argument that needs to be updated in the recursive call. In Haskell, we

```
data Unit a            =  E
type Tr t a            =  (t a, a, t a)
data Red t a           =  C (t a)  |  R (Tr t a)
data AddLayer t a      =  B (Tr (Red t) a)
data RB t a            =  Base (t a)  |  Next (RB (AddLayer t) a)
type Tree a            =  RB Unit a

member :: Ord a ⇒ a → Tree a → Bool
member x t             =  rbmember x t (\ _ → False)

rbmember :: Ord a ⇒ a → RB t a → (t a → Bool) → Bool
rbmember x (Base t) m  =  m t
rbmember x (Next u) m  =  rbmember x u (bmem x m)

bmem :: Ord a ⇒ a → (t a → Bool) → AddLayer t a → Bool
bmem x m (B(l, y, r))
  | x < y              =  rmem x m l
  | x > y              =  rmem x m r
  | otherwise          =  True
rmem :: Ord a ⇒ a → (t a → Bool) → Red t a → Bool
rmem x m (C t)         =  m t
rmem x m (R(l, y, r))
  | x < y              =  m l
  | x > y              =  m r
  | otherwise          =  True
```

Fig. 2. Proper red-black trees.

can hide this argument from view by using the class system (Hinze, 1999a), but that is merely a matter of presentation.

The penalty for implementations using this data structure will necessarily contain:

- the cost for the indirections *C t*;
- the cost for passing through (or maintaining) the *Next* constructors;
- the cost of the dictionary updates (the mentioned functional argument).

A black coloured tree can either have red or black coloured subtrees — this is the reason for the two constructors at type *Red*, and in particular for the extra overhead caused by the constructor *C* as mentioned in the first point. The purpose of *C* is to embed black coloured trees into the type of potentially red coloured trees which are exactly the kind of trees permitted as subtrees of black nodes. In other words, any application of *C* is overhead, we pay for the typing. In the following I shall put this point aside as it is completely independent from the other issues — it should just be mentioned though that this particular subproblem can be solved through the use of so-called refinement types (Davies, 1997).

The other two points incur costs proportional to the height of the tree. Also, search, insertion and deletion for this data structure operate in time proportional to the tree height – implementations are provided on the JFP home page

`http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html`

```
type Tr t a b                     =  (t a b, a, t a b)
data Red t a b                    =  C (t a b) | R (Tr t a b)
data Black a b                    =  E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] → a → Red Black a [b] → Red Black a b
balanceL (R(R(a, x, b), y, c)) z d  =  R(B(C a, x, C b), y, B(c, z, d))
balanceL (R(a, x, R(b, y, c))) z d  =  R(B(a, x, C b), y, B(C c, z, d))
balanceL (R(C a, x, C b)) z d       =  C(B(R(a, x, b), z, d))
balanceL (C a) x b                  =  C(B(a, x, b))
```

Fig. 3. Top-down typing.

In other words, the penalty slows the algorithms down by a constant factor. Can we avoid these costs?

2 Employing existential types

We can indeed reduce the performance penalty, by exploiting a language extension supported by most (if not all) Haskell compilers.

Figure 3 shows another type definition for red-black trees that again uses nested datatypes, i.e. one argument of a type constructor changes during recursion – the last argument for type constructors *Red* and *Black*. However, this argument is a *phantom type*, it is not used anywhere, no data component has that type.

I have not invented phantom types. Erik Meijer and Daan Leijen seem to be using them regularly in their work, e.g. in Finne *et al.* (1999) to express inheritance. However, in our application the phantom type is even more elusive – it does not interfere with the code, its only purpose is to make the *type checker* check the tree balancing. We simply record in this argument the depth of the node in the tree, i.e. how many black levels we have passed from the root of the tree.

As no data component uses this changing type, the code for the program is identical to a much more relaxedly typed version which ensured the colouring but not the balancing – only the type annotations change. One such example (of unaffected code) is the *balanceL* operation in figure 3. Its type tells us that its first tree argument is (potentially) infrared, the second (potentially) red and that the result is (potentially) red. Moreover, the depth of the result is one less than the depths of the two others: this is recorded in the last type argument.

The *depth* of the trees (relative to some other tree) is not quite what we want, we need to reason about their *heights*. However, subtrees at the same depth k in a balanced tree of height n have necessarily the same height $(n-k)$. The function *balanceL* creates a tree we are allowed to place at depth $k-1$. We really need that this tree has height $n-k+1$ – it does, but this is not enforced through the types alone. We also need to restrict the use of the polymorphic E constructor.

Figure 4 shows the main part of the insertion algorithm (I omitted *balanceR* which is completely dual to *balanceL*). Insertion into black and red coloured nodes has been split as they now have different types. Again, the types tell us about both the colouring and the depth, e.g. inserting into a (potentially) red tree gives us a (potentially) infrared tree of the same depth.

```
insB :: Ord a ⇒ a → Black a b → Red a b
insB x E              =  R(E, x, E)
insB x t@(B(a, y, b))
      | x < y         =  balanceL (insR x a) y b
      | x > y         =  balanceR a y (insR x b)
      | otherwise     =  C t

insR :: Ord a ⇒ a → Red a b → Red (Red Black) a b
insR x (C t)          =  C(insB x t)
insR x t@(R(a, y, b))
      | x < y         =  R(insB x a , y , C b)
      | x > y         =  R(C a , y , insB x b)
      | otherwise     =  C t
```

Fig. 4. Insert.

```
newtype Tree a       =   forall b . ENC (Black a b)
empty                =   ENC E

insert               ::  Ord a ⇒ a → Tree a → Tree a
insert x (ENC t)     =   ENC(blacken(insB x t))

blacken              ::  Red Black a b → Black a b
blacken (C u)        =   u
blacken (R(a, x, b)) =   B(C(inc a) , x , C(inc b))
```

Fig. 5. Existential type.

The algorithm is wrapped up in figure 5. In order to keep operating with tree depths in a safe manner (i.e. using depths as a reliable source of information for their heights) we have to keep the depths of differently constructed trees separate. This is achieved by using a fresh existential type variable[2] for every freshly-built tree.

The figure also hints at the only computational overhead required for this version, the calls of the function *inc* inside the definition of *blacken*. We need to call this operation whenever the height of the overall tree increases. In that case, the top node changes its colour from red to black and thus the depth of every single node in the tree goes up by one. Although the tree itself does not change (*inc* really is the identity function, see figure 6), the type system forces us to traverse the entire tree. This linear cost arises only when the height increases which happens with a probability of $(\log n)/n$ inserting a random element into a random tree of size n. Thus, the expected costs are still $O(\log n)$; this approximation still applies under strict evaluation, provided we are prepared to live with the worst-case cost of $O(n)$. Only a few pathological usage patterns can make the amortised costs (Okasaki, 1998) exceed that bound, e.g. when we repeatedly delete/insert while the tree is at the borderline of a certain height. The situation does not change under strict evaluation, unless we make the tree constructors non-strict — in which case single-threadedness becomes an additional worry.

[2] Syntax for this feature varies between compilers as it is non-standard.

```
inc                 :: Black a b → Black a [b]
inc                  = tickB

tickB               :: Black a b → Black a c
tickB E              = E
tickB (B(a, x, b))   = B(tickR a , x , tickR b)

tickR               :: Red a b → Red a c
tickR (C t)          = C(tickB t)
tickR (R(a, x, b))   = R(tickB a , x , tickB b)
```

Fig. 6. Depth adjustment.

3 Deletion

Deletion of elements is a more intricate operation. Notice that the auxiliary *insB* function of the insertion algorithm maintains the property that both its argument and result have the same height. Deletion cannot maintain the same invariant, for a very simple reason: if we have a singleton black tree and delete its sole element then the only possible outcome is the empty tree – and this already reduces the height. More generally, if we (successfully) delete an element from any tree without red-coloured nodes then the height of the tree has to be reduced; singleton black trees are just a special case.

We can maintain a different invariant though. Whenever we attempt to delete something from a *black tree* of height $n+1$ we return a tree of height n, while deletion from red trees (and the empty tree) preserves the height. This is even possible if the deletion attempt fails as we can always redden the root node, again permitting infrared trees. Overall, this is a slight improvement over the deletion algorithms in Hinze (1998) and Reade (1992), which represent deletion underflow in the data rather than putting it into the structure of the algorithm.

The full algorithm can be found on the JFP web site. While Hinze's algorithm essentially tries to mimic the traditional imperative algorithm, my version is closer to Reade's as it is also based on a recursive *append* operation. The more complicated structure of the deletion algorithm is bad news for the *higher-order nested* version of red-black trees (from figure 2), because it needs to update that structure whenever tree heights change. In both Hinze's and my algorithm (when adapted to that type) this means updating a class dictionary with at least two functions, significantly increasing the computational overhead.

Of more interest for this paper though is how the algorithm interfaces with the existential type variables, see figure 7. The function *delB* is the dual to *insB*, it removes an element from a black tree. The result is a potentially infrared tree of depth 1. If that tree is either red or infrared (first two cases) we simply blacken the top red node and thus obtain the required black tree of depth 0. In the third case the returned tree is already black and it is here where we have a deletion underflow – the height of the tree decreases. However, in contrast to insertion, we do not need to adjust the types in this case as we can abstract any type we like when we

```
delete :: Ord a ⇒ a → Tree a → Tree a
delete x (ENC t)        =
    case delB x t of
        R p             →   ENC (B p)
        C(R(a, x, b))   →   ENC(B(C a, x, C b))
        C(C q)          →   ENC q
delB :: Ord a ⇒ a → Black a b → Red (Red Black) a [b]
...
```

Fig. 7. Deletion wrap-up.

introduce an existential type variable. Although *q* has type *Black a* [*b*] in that last case, *ENC q* still type-checks.

Therefore, this deletion operation has no computational overhead whatsoever for the type discipline that enforces balancing, unless we count the part responsible for the colour discipline.

4 Conclusion

We know how we can maintain the invariants of red-black trees through Haskell's type system, using nested datatypes. This causes a small but noticeable overhead. Most of this overhead can be removed by the clever use of existential types.

One can also easily eliminate all the checks once correctness is established: just eliminate the phantom type parameter we used to pass on the existential type. This removes both polymorphic recursion and existentials, but leaves the algorithm virtually unchanged – boosting the performance slightly as *inc* can be replaced by the identity.

While the implementation has practical advantages over higher-order nested types, it is less clear how it would compare to a dependently typed version, in particular Hongwei Xi's implementation of red-black trees in de Caml (Xi, 1999). Xi's version also avoids the costly higher-order parameters required by the higher-order nested types, but it is not quite clear to me how much (if anything) from the type system invades the run-time system.

References

Bayer, R. (1972) Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta informatica*, **1**, 290–306.

Cormen, T. H., Leiserson, C. E. and Rivett, R. L. (1990) *Introduction to Algorithms.* MIT Press.

Davies, R. (1997) *A refinement-type checker for Standard ML.* AMAST'97 presentation.

Finne, S., Daan, L., Meijer, E. and Peyton Jones, S. (1999) Calling heaven from hell and hell from heaven. *Proceedings ACM SIGPLAN International Conference on Functional Programming*, pp. 114–125. ACM Press.

Hinze, R. (1998) *Numerical representations as higher-order nested datatypes.* Technical report IAI-TR-98-12, Universität Bonn.

Hinze, R. (1999a) Constructing red-black trees. *Workshop on algorithmic aspects of advanced programming languages*, pp. 89–99.

Hinze, R. (1999b) Manufacturing datatypes. *Workshop on Algorithmic Aspects of Advanced Programming Languages*, pp. 1–16.

Kahrs, S. (1996) Limits of ML-definability. *Proceedings of PLILP'96: Lecture Notes in Computer Science 1140*, pp. 17–31. Springer-Verlag.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Okasaki, C. (1999) Functional pearl: Red-black trees in a functional setting. *J. Functional Programming*, **9**(4), 471–477.

Reade, C. (1992) Balanced trees with removals, an exercise in rewriting and proof. *Science of Computer Programming*, **18**(2), 181–204.

Sedgewick, R. (1988) *Algorithms.* Addison-Wesley.

Xi, H. (1999) Dependently typed data structures. *Workshop on Algorithmic Aspects of Advanced Programming Languages*, pp. 17–32.