RED AND BLACK TREES

kanika khurana, Mansi verma, Gaurav kumar
Department of Electronics and Communication Engineering
Dronacharya College of Engineering, Gurgaon, India

***Abstract*- I present a new implementation of balanced binary search trees, compatible with the MSets interface of the Coq Standard Library. Like the current Library implementation, mine is formally verified (in Coq) to be correct with respect to the MSets specification, and to be balanced (which implies asymptotic efficiency guarantees). Benchmarks show that my implementation runs significantly faster than the library implementation, because (1) Red-Black trees avoid the significant overhead of arithmetic incurred by AVL trees for balancing computations; (2) a specialized delete-min operation makes priority-queue operations much faster; and (3) dynamically choosing between three algorithms for set union/intersection leads to better asymptotic efficiency.**

I. INTRODUCTION

An important and growing body of formally verified software (with machine-checked proofs) is written in pure functional languages that are embedded in logics and theorem provers; this is because such languages have tractable proof theories that greatly eases the verification task. Examples of such languages are ML (embedded in Isabelle/HOL) and Gallina (embedded in Coq). These embedded pure functional languages extract to ML that can be compiled with optimizing compilers, so it's not crazy to think of building real software this way that's efficient enough to solve real problems. Balanced binary search trees are an important data structure in computer science, and particularly so in pure functional programming. They are used to implement the abstract type of sets over totally ordered keys, with O(logN) insertion, lookup, and deletion. In a programming language with a sufficiently powerful module system (MacQueen, 1990) such as that of Standard ML or OCaml, one can specify the interface of the set abstract data type, parametrized over another abstract data type of totally ordered keys. Filliˆatre and Letouzey (Filliˆatre & Letouzey, 2004) show that in the Coq proof assistant, one can go even farther: in the keys module are not only the comparison operations on keys, but the specification expressed in logic (the Calculus of Inductive Constructions) that the keycomparison really is totally ordered; and in the sets module are the logical correctness specifications of all of the operations, also expressed in logic.

II. WHY ARE AVL TREES SLOW?

Integer arithmetic in the Coq standard library is constructed from inductive structures as follows. **Inductive** positive := xI : positive ! positive | xO : positive ! positive | xH : positive. **Inductive** Z := Z0 : Z | Zpos : positive ! Z | Zneg : positive ! Z. A positive number is either 1, represented by xH, or 2n, represented by xO(n), or 1+2n, represented by xI(n). Whenever we reason about integers (type Z) in Coq, we are in fact reasoning about data structures such as (Zneg(xI(xO(xH)))) and (Zpos(xO(xI(xI(xH))))). Such reasoning takes time (typically) linear in the size of the data structure, and logarithmic in the size of the numbers represented. This explains why the implementation of AVL trees in the Coq library performs slowly for insert: balance numbers are represented as Z, and thus there is a logN penalty; effectively insert k t takes time log2 |t|. One might think, "when extracting to ML programs, why can't we represent Z as a single-word native integer, and do machine-native arithetic?" Indeed, Filliˆatre and Letouzey 2 One can do Red-Black trees with 0 bits of balance information, at the cost of extra comparisons Ex. 13.65, p. 560. But I want the data structure to be efficient even in regimes where comparisons are expensive, so this technique is not attractive.However, I believe they are underestimating a significant problem: machine arithmetic arithmetic can overflow. If one axiomatizes this overflow then one has many proof obligations of the form x+y<231. In practice, these proof obligations are overwhelmingly nasty.

Furthermore, the specifications themselves would get much more complicated. One of themost important methods by which people have made progress in verified algorithms is by the clever trick of using infinite-precision integers, not because they will ever overflow, but so that the proofs are simpler. Theorem: If we were to use machine integers to store the balance information for AVL

trees, those integers would never overflow. Proof. The height stored in an AVL tree never exceeds the log of the number of pointers in the tree, and thus on any machine where integers are at least as large as pointers, the height
of the tree is representable.
It is exceedingly difficult to convert this theorem to a machine-checkable result, and
I will not even try. Thus, one can see why Filliˆatre and Letouzey did not attempt using
fixed-precision arithmetic for heights of AVL trees.
But there's a simpler way. One should simply use a representation of balanced search
trees that does not require integers: Red-Black trees.

III. LEFT-LEANING RED-BLACK TREES.

Sedgewick (Sedgewick, 2008) proposed left-leaning Red-
Black trees, a data structure identical to ordinary Red-Black trees but with the extra constraint
that no node has a red left child. This reduces the number of cases to be handled,
either in the (imperative, pointer-swizzling) implementation of the algorithm or the proofs
of correctness and balance.
In addition, Sedgewick shows how to factor the implementation of rebalancing Red-
Black trees into three operations, rotateLeft, rotateRight, and colorFlip; the proofs can be
refactored correspondingly.
My student Max Rosmarin (Rosmarin, 2011) studied the question of whether using the
left-leaning invariant would mix well with the Okasaki-style functional program, so as
to factor the implementations and proofs. Rosmarin demonstrated that Okasaki's balance
function can be factored into Sedgewick's three operations. Although it is not conceptually
more complex, the factored function has more lines of code. Recall that Okasaki's function,
as I presented it here, has only 10 lines, which is hard to improve on.
The proofs can be factored as well. Recall that my proofs about Okasaki's balance
function took 1125 steps. Undoubtedly, proofs factored in left-leaning style would take
fewer steps. But my 1125 steps were computed automatically from the 8 one-line proof
tactics outlined in **Ltac** do balance. In that sense, my proof is "routine." Rosmarin found
that left-leaning factored proofs were not as "routine," and therefore required more human
effort to build.

IV. DYNAMICALLY CHOOSING BETWEEN THE IMPLEMENTATIONS.

To measure whether |s| _ |t| or
|t|_|s| or neither, one does not want to compute |s|, which takes linear time. But we can
cheaply compute the approximate log of |s|, that is, the black-node height of the tree (since
the black-node depth of every leaf is the same). Even more cheaply, we can test whether
the black-height of s is at least twice the black-height of t, or vice-versa.
Figure 1. Induction scheme for treeify
Lemma treeify f length:
8n l, length l > nat of P n ! length(snd(treeify f n l))+nat of P n = length l.
Lemma treeify g length:
8n l, length l _ nat of P n ! length(snd(treeify g n l))+nat of P n = S (length l).
Lemma treeify g induc:
8fP gP : positive ! list key ! tree _ list key ! Prop,
(_1_) (8 l n' t1 x l2 t2 l3,
length l >= nat of P n' ! length l2 >= nat of P n' ! fP n' l (t1, x::l2) !
treeify f n' l = (t1, x::l2) ! fP n' l2 (t2, l3) ! treeify f n' l2 = (t2, l3) !
fP (xI n') l (T Black t1 x t2, l3)) !
(_2_) (8 l n' t1 x l2 t2 l3,
length l >= nat of P n' ! S(length l2) >= nat of P n' ! fP n' l (t1, x :: l2) !
treeify f n' l = (t1, x :: l2) ! gP n' l2 (t2, l3) ! treeify g n' l2 = (t2, l3) !
fP (xO n') l (T Black t1 x t2, l3)) !
(_3_) (8 x l1, fP xH (x :: l1) (T Red E x E, l1)) !
(_4_) (8 l n' t1 x l2 t2 l3,
length l >= nat of P n' ! S(length l2) >= nat of P n' ! fP n' l (t1, x :: l2) !
treeify f n' l = (t1, x :: l2) ! gP n' l2 (t2, l3) ! treeify g n' l2 = (t2, l3) !
gP (xI n') l (T Black t1 x t2, l3)) !
(_5_) (8 l n' t1 x l2 t2 l3,
S(length l) >= nat of P n' ! S(length l2) >= nat of P n' ! gP n' l (t1, x :: l2) !
treeify g n' l = (t1, x :: l2) ! gP n' l2 (t2, l3) ! treeify g n' l2 = (t2, l3) !
gP (xO n') l (T Black t1 x t2, l3)) !
(_6_) (8 l, gP xH l (E, l)) !
(_ conclusion _) 8n l, length l >= nat of P n ! gP n l (treeify g n l).

Definition skip red t := **match** t **with** T Red t') t' |) t **end**.
Definition skip black t := **match** skip red t **with** T Black t') t' | t') t' **end**.
Fixpoint compare height (sx s t tx: tree) : comparison :=
match skip red sx, skip red s, skip red t, skip red tx **with**
| T sx' , T s' , T t' , T tx') compare height (skip black tx') s' t' (skip black tx')
| , E, , T) Lt
| T , , E,) Gt
| T sx' , T s' , T t' , E) compare height (skip black sx') s' t' E
| E, T s' , T t' , T tx') compare height E s' t' (skip black tx')
| , , ,) Eq
end.

The calculation compare height s s t t starts the pointers sx,tx racing down the two trees at double-speed, and the pointers s,t walking down at normal speed. Depending on which of these four pointers bottoms out first, we can say informally that

c1 log |s| <
1
2
log |t|, c2 log |s| < log |t|^log |s| > c2 log |t|, or
1
2
log |s| > c3 log |t|

for various constants ci close to 1.

We do not have to prove this formally! The compare height function will be used only to select between three different proved-correct implementations of set-union. If we get it wrong, the algorithm will still be formally verified as functionally correct, but it may be inefficient. For efficiency we are relying on a combination of formal proofs about balance properties, plus informal proofs about efficiency. The informal proof is simple.

Theorem: compare height is correct.
Proof: Obviously it's correct.

Then we combine the three versions of set-union, as follows:

Definition union (s t: tree) : tree :=
match compare height s s t t **with**
| Lt) fold insert s t
| Gt) fold insert t s
| Eq) linear union s t
end.

where fold is a function such that (for example),
fold insert s t = insert s1 (insert s2 (insert s3 . . . (insert sn t) . . .))
where si are all the keys in tree s.

V. CONCLUSION

Balanced binary search trees are an important data structure, especially for pure functional programming and therefore for verified software. However, several design decisions influence the efficiency of search-tree algorithms. In particular, because in Coq the use of arithmetic usually imposes a logN penalty, it is advantageous to use search-tree algorithms that avoid arithmetic as they rebalance trees. In addition, for use as priority queues a specialize delete-min operation is much more efficient than separate min-elt and delete; and one can speed up set union or intersection by dynamic choice of algorithm depending on the relative depths of the trees.

Acknowledgments. This research was supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).

REFERENCES

Armand, Micha¨el, Gr´egoire, Benjamin, Spiwack, Arnaud, & Th´ery, Laurent. (2010). Extending Coq with imperative features and its application to SAT verification. Pages 83–98 of: Itp'10: International conference on interactive theorem proving, vol. LNCS 6172. Springer.

Filliˆatre, J.-C., & Letouzey, P. (2004). Functors for Proofs and Programs. Pages 370–384 of: Esop'04: European symposium on programming, vol. LNCS 2986. Springer.

Hinze, Ralf. 1999 (Sept.). Constructing red-black trees. Pages 89–99 of: Okasaki, Chris (ed), Waaapl'99: Workshop on algorithmic aspects of advanced programming languages.

Kahrs, Stefan. (2001). Red-black trees with types. Journal of functional programming, 11(04), 425–432.

MacQueen, David B. (1990). A higher-order type system for functional programming. Pages 353–68 of: Research topics in functional programming. Reading, MA: Addison-Wesley.

Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avraham, Govereau, Paul, & Birkedal, Lars. 2008

(Sept.). Ynot: Dependent types for imperative programs. Icfp ’08: Proceedings of the 13th acm sigplan international conference on functional programming.

Okasaki, Chris. (1999). Red-black trees in a functional setting. J. functional programming, 9(4), 471–477.

Rosmarin, Max. 2011 (Aug.). Red-black trees in a functional context: Left-leaning and otherwise.

Princeton University Department of Computer Science.

Sedgewick, Robert. (2008). Left-leaning red-black trees.