Formally Verified Low-Level C Implementation of Crit-Bit Trees in a Live Verification Tool

Viktor Fukala

MIT, USA and D-INFK, ETH Zurich, Switzerland

This extended abstract was accepted into the Student Research Competition at PLDI'24. This version has been updated according to reviewers' feedback.

1 ABSTRACT

Using a recent live verification tool [7], we implement a crit-bit tree in C syntax and verify the implementation against the Bedrock2 C-like semantics. This is the first formally verified implementation of crit-bit trees, the first implementation of an efficient key-value store in Bedrock2, and one of only a limited number of formally verified, low-level, imperative implementations of a map data structure in general. Our work is the first significant case study of [7]. In our benchmarks for lookup, insertion, and deletion, our crit-bit tree implementation was at most 22 % slower than `std::map` in libstdc++.

2 PROBLEM AND MOTIVATION

A crit-bit tree [1, 5] is a little-known tree data structure that supports efficient execution of ordered dictionary operations (lookup, insert, erase, ordered predecessor/successor lookup). While providing functionality similar to the more popular alternatives like AVL or red-black trees, it requires no rebalancing mechanism. In this project, we implemented a crit-bit tree in C and formally verified our implementation with respect to the C-like semantics of Bedrock2 [6]. For both implementation and verification, we used a prototype of a live verification tool [7] that is being developed as part of the Bedrock2 ecosystem.

We contribute

- the first efficient implementation of a key-value store ready for use in other Bedrock2 projects,
- insights into the verification of crit-bit trees in general,
- one of a limited number of verified, imperative, low-level implementations of data structures with map functionality
- the first significant case study on the usability of the new live verification tool.

3 BACKGROUND AND RELATED WORK

There are many existing formally verified implementations of tree data structures in functional languages, e.g. [2, 3]. These tend to be simpler than imperative implementations due to the similarity of the proof assistant and the implementation languages. Our low-level, imperative implementation has the advantages of: interoperability with other low-level code, more complete correctness guarantees in practice when the lower SW/HW layers are also verified (as in [6]), and the potential for better performance.

Some projects proceed by first verifying a functional implementation and then refining it to an imperative one, e.g. [11]. In contrast, we decided to verify our imperative implementation directly and we tend to believe that it led to a lower overall proof burden.

In automated program verifiers [4], verification failures are often difficult to debug because the verifier doesn't show an intuitive representation of the proof state from which it couldn't progress. Further, program implementation and its proof are usually separated making editing one or the other cumbersome. The live verification tool addresses both these issues and our project is the first significant work to use the tool's program verification approach.

3.1 Crit-Bit Trees

In our crit-bit trees (CBTs), both the keys and values are from the set of all bitstrings of a fixed bitwidth BW, $\mathcal{K} = \mathcal{V} = \{\texttt{false}, \texttt{true}\}^{BW}$.

A non-empty CBT is a binary tree. In contrast to binary search trees, only the leaf nodes store key-value pairs. Each of the other, non-leaf, nodes stores an index $i \in \{0, \ldots, BW-1\}$ of a *critical bit* (hence *crit-bit* trees). This signifies that in the subtree below such a node, the bitstrings of some keys have `false` at index i (those are required to be in the left subtree) while others have `true` (those must be right). The only other invariant is that a node can only store a critical bit index that is greater than that of its parent.

Efficiency-wise, CBTs are an improvement over traditional tries as they don't have an internal node for all bits but only for the critical ones (when both `false` and `true` are possible). They are also quite similar to, yet simpler than, PATRICIA tries [10].

A good overview of CBTs can be found in [1, 5] and similar data structures are also described in [8, 12]. Noticeably, CBTs don't require any restructuring (rebalancing) operations which would both lead to variations in running time and complicate our proofs. All operations (lookup, insert, delete, find next with respect to lexicographic order of bitstrings) can be implemented with running time in $O(BW) = O(\log|\mathcal{K}|)$; but in practice, the number of critical bits can be much lower than BW and execution therefore faster.

3.2 Bedrock2 and Live Verification Tool

Using Coq, the Bedrock2 project develops new methods for verifying low-level systems applications. It includes the formalization and verification of an entire SW/HW stack from the hardware design of a RISC-V processor and the semantics of its instruction set to a corresponding compiler for a programming language with C-like semantics. As part of Bedrock2, the live verification tool aims to significantly improve on the end of writing the top-level programs. When implementing a program, after each statement the user enters, the tool displays the current symbolic state and any potential proof obligations that couldn't be discharged automatically. The user can then prove these manually inline without abandoning the unfinished implementation. Thanks to a set of Coq notations, individual statements of the program are written in familiar C syntax such that the final Coq file can even be directly compiled by a production C compiler like GCC.

4 APPROACH AND UNIQUENESS

4.1 Prefixes of Keys

Much of our proofs relies on efficient reasoning about prefixes of keys. In a valid CBT, each node, explicitly storing a critical bit index $i \in \{0, \ldots, BW-1\}$, also has an implicitly associated prefix of length i, which is the longest common prefix of all keys in its subtree. We observed that it was practical to define prefixes as a standalone concept to streamline our proofs. We denote all the potential prefixes $\mathcal{P} = \{\texttt{false}, \texttt{true}\}^{\leq BW}$. $\mathcal{P}$ exhibits a lot of useful structure: there is the canonical injection $\iota : \mathcal{K} \to \mathcal{P}$, a relation $\texttt{pfx\_le} \subseteq \mathcal{P} \times \mathcal{P}$ with p_1 $\texttt{pfx\_le}$ $p_2 \iff$ "p_1 is a prefix of p_2," an operation $\texttt{pfx\_meet} : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ with $\texttt{pfx\_meet}(p_1, p_2)$ equal to the longest common prefix of p_1 and p_2, and a function $\texttt{pfx\_len} : \mathcal{P} \to \mathbb{N}$ for the length of a prefix. These definitions are useful for expressing assertions in our CBT proofs (e.g., $\texttt{pfx\_len}(\texttt{pfx\_meet}(\iota(k_1), \iota(k_2)))$ is the critical bit index of two keys) and at the same time, they relate to familiar mathematical structures $((\mathcal{P}, \texttt{pfx\_le})$ is a partially-ordered set, $(\mathcal{P}, \texttt{pfx\_le}, \texttt{pfx\_meet})$ is a meet-semilattice). Finally, for a map M, $\emptyset \neq M \subseteq \mathcal{K} \times \mathcal{V}$, we defined $\texttt{pfx\_mmeet}(M)$ as the meet of all $\iota(k)$ for $(k, v) \in M$ (order does not matter because `pfx_meet` is associative and commutative). With this, $\texttt{pfx\_mmeet}(M)$ is the longest common prefix of all keys in M.

By using these definitions and by exploiting the general properties of a semilattice, we can often avoid the low-level manipulation of prefix representations in the correctness proofs of our CBT functions. The abstract prefix definitions make it easier (both in manual proofs and in automated proof scripts) to focus on the crucial information and to understand why specific assertions do (not) hold.

4.2 CBT Predicate

Central to the verification is the predicate which asserts that a pointer points to a valid CBT representing a given map c (c for *content*). Unmodified, we show this predicate for non-empty CBTs:

```
Fixpoint cbt' (tree: tree_skeleton)
  (c: word_map) (a: word): mem -> Prop :=
  match tree with
  | Leaf => EX k v,
      <{ * emp (a <> /[0])
         * freeable ltac:(wsize3) a
         * <{ + uintptr /[ltac:(bw)]
              + uintptr k
              + uintptr v }> a
         * emp (c = map.singleton k v) }>
  | Node treeL treeR => EX (aL: word) (aR: word),
      <{ * emp (a <> /[0])
         * freeable ltac:(wsize3) a
         * <{ + uintptr /[pfx_len (pfx_mmeet c)]
              + uintptr aL
              + uintptr aR }> a
         * cbt' treeL (half_subcontent c false) aL
         * cbt' treeR (half_subcontent c true) aR }>
  end.
```

Here,`<{ * _ ... * _ }>` is the separating conjunction of several memory predicates, `<{ + _ ... + _ }>` combines predicates with contiguous memory footprints of known sizes into a predicate that asserts that the constituents appear at a particular address one after the other contiguously in memory.

5 RESULTS

Statistics of our verified implementation:[1]

total LOC	LOC in C	# of C funcs	verification time
5495	356	23	8 min. 40 s

Out of the 23 C functions, 14 are internal helper functions and 9 are meant to be exposed. Those are `cbt_init`, `cbt_lookup`, `cbt_insert`, `cbt_delete`, `cbt_get_min`, `cbt_get_max`, `cbt_next_ge`, `cbt_next_gt`, and `page_from_cbt`. `get_min`, `get_max`, and `next_*` return a key-value pair with a particular property related to the ordering of the keys when interpreted as unsigned integers: `get_min` / `get_max` return the key-value pair with the min. / max. key in the CBT; `next_ge` / `next_gt` return the pair with the smallest key greater equal / greater than a specific key given as argument (this key need not be in the CBT) and they can be used to iterate over ranges of entries in a CBT. `page_from_cbt` demonstrates the iterator capabilities: given arguments n and k, it reads the next n key-value pairs at or after key k from a CBT into an array.

Apart from abstracting away the representation of prefixes, another advantage of our prefix formalization from 4.1 is that (e.g., in the CBT predicate 4.2) we can express a node's prefix explicitly as `pfx_mmeet c` and can avoid assertions with existentially quantified prefixes, which we used to have in our proofs initially. It was generally easier to prove a fully instantiated assertion about a property of `pfx_mmeet` than an assertion where we first had to find the right prefix to instantiate an existential quantifier with.

5.1 Implementation Performance

We insert $N = 2^{20}$ key-value pairs with pseudorandom keys ($\in [0, N)$) into an empty container (**insert**). Then we perform either an in-order iteration over the entire container (**iter**), or a lookup (**lookup**) or deletion (**delete**) of another N pseudorandom keys.[2]

	insert	**iter**	**lookup**	**delete**
CBT	307 ms	90 ms	316 ms	254 ms
`std::map` (red-black tree)	252 ms	44 ms	295 ms	288 ms
`std::unordered_map`	50 ms	-	32 ms	33 ms

We compare with `std::map` because that is the go-to standard library alternative that provies the functionality of our CBT. For lookup, insertion, and deletion, we measured our implementation to be no more than 22 % slower than `std::map`. For in-order iteration, our CBT is around two time slower than `std::map`, which we attribute to our unoptimized implementation of CBT iteration.

[1] Code accessible at https://github.com/vfukala/bedrock2/tree/iterators.

[2] Using g++ version 13.2.0 with `-O2` on Intel Core i9-12900H. For a fairer comparison, the results we show are from running with the mimalloc memory allocator [9], which, compared to the standard memory allocator, reduced the execution times especially for `std::map`.

6 ACKNOWLEDGMENTS

I would like to thank Samuel Gruetter, Adam Chlipala, and MIT UROP for their support, advice, and funding.

REFERENCES

[1] Adam Langley. Crit-bit trees. https://www.imperialviolet.org/binary/critbit.pdf.
[2] Appel, A. W. Efficient verified red-black trees. https://www.cs.princeton.edu/~appel/papers/redblack.pdf.
[3] Appel, A. W., and Leroy, X. Efficient extensional binary tries. In *Journal of Automated Reasoning* (2023).
[4] Chlipala, A. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, Association for Computing Machinery, p. 234–245.
[5] Daniel J. Bernstein. https://cr.yp.to/critbit.html.
[6] Erbsen, A., Gruetter, S., Choi, J., Wood, C., and Chlipala, A. Integration verification across software and hardware for a simple embedded system. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, p. 604–619.
[7] Gruetter, S., Fukala, V., and Chlipala, A. Live verification in an interactive proof assistant. *Proc. ACM Program. Lang. 8*, PLDI (jun 2024).
[8] Gwehenberger, G. Use of a binary tree structure for processing files. *Elektron. Rechenanlagen 10*, 5 (1968), 223–226.
[9] mimalloc contributors. https://github.com/microsoft/mimalloc.
[10] Morrison, D. R. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4 (oct 1968), 514–534.
[11] Mündler, N., and Nipkow, T. A verified implementation of b+-trees in isabelle/hol. In *Theoretical Aspects of Computing – ICTAC 2022* (Cham, 2022), Springer International Publishing, pp. 324–341.
[12] Sklower, K. A tree-based packet routing table for berkeley unix. In *USENIX Winter* (1991).