# Efficient Extensional Binary Tries

Andrew W Appel, Xavier Leroy

**Noname manuscript No.**
(will be inserted by the editor)

# Efficient Extensional Binary Tries

**Andrew W. Appel · Xavier Leroy**

the date of receipt and acceptance should be inserted later

**Abstract** Lookup tables (finite maps) are a ubiquitous data structure. In pure functional languages they are best represented using trees instead of hash tables. In pure functional languages within constructive logic, without a primitive integer type, they are well represented using binary tries instead of search trees.

In this work, we introduce *canonical binary tries*, an improved binary-trie data structure that enjoys a natural extensionality property, quite useful in proofs, and supports sparseness more efficiently. We provide full proofs of correctness in Coq.

We provide microbenchmark measurements of canonical binary tries versus several other data structures for finite maps, in a variety of application contexts; as well as measurement of canonical versus original tries in a big, real system. The application context of *data structures contained in theorem statements* imposes unusual requirements for which canonical tries are particularly well suited.

## 1 Introduction

Lookup tables—finite maps from identifiers to bindings—are a central data structure in many kinds of programs. We are particularly interested in programs that are proved correct—compilers, static analyzers, program verifiers. Those programs are often written in pure functional languages, since the proof theory of functional programming is more tractable than those of imperative or object-oriented languages. Thus we focus on applications written in the functional languages internal to the logics of Coq or HOL. Such programs *may* be "extracted" to programming languages such as OCaml, Standard ML, or Haskell, and compiled with optimizing compilers for those languages; the CompCert C compiler [8] and

**Draft of October 11, 2021.** There are no conflicts of interest. Code is available at `https://github.com/xavierleroy/canonical-binary-tries`.

Andrew W. Appel
Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ, 08540
E-mail: appel@princeton.edu

Xavier Leroy
Collège de France, 3 rue d'Ulm, 75005 Paris
E-mail: xavier.leroy@college-de-france.fr

the Verasco static analyzer [7] are examples. On the other hand, some programs (such as the Verified Software Toolchain program verifier [1]) don't use extraction: those programs are reduced within the logic of a proof assistant, so that the program verifier can gracefully interact with user-driven interactive proof in the ambient logic.

Many proved-correct programs of this kind deliberately use an improverished set of primitives, so that fewer axioms need to be assumed (or equivalently, fewer lines of proof-checker kernel code need to be trusted). In particular, there may be no primitive integer type implemented in a machine word. Instead, integers (or whatever type is used for *keys,* the domain of the finite maps) may be constructed from the inductive data types of the logic. Consequently, we cannot assume that a less-than comparison of two keys takes constant time; it will be logarithmic in the magnitude of the numbers.

These criteria led to a preliminary design for a data structure and its algebraic interface: binary tries, also called positive trees (for reasons that will become clear). This data structure was used in the CompCert C compiler, the Verified Software Toolchain (VST), the Verasco verified static analyzer, and in other applications. In the process, we have learned to demand more and more criteria of efficiency and provability from them:

*Core Requirements*

1. Basic operators: empty (the finite map with an empty domain), get (the lookup operator), and set (the update operator).
2. Proofs[1] of basic laws: $\mathsf{get}\ i\ \mathsf{empty} = \mathsf{None}$, $\mathsf{get}\ i\ (\mathsf{set}\ i\ v\ m) = \mathsf{Some}\ v$, $i \neq j\ \rightarrow \mathsf{get}\ i\ (\mathsf{set}\ j\ v\ m) = \mathsf{get}\ i\ m$.
3. A purely functional implementation, which rules out (for example) hash tables.
4. A *persistent* semantics, so that $\mathsf{set}\ j\ v\ m$ does not destroy $m$; this arises naturally from the purely functional implementation.
5. Efficient representation of keys.
6. Efficient asymptotic time complexity of get and set operations, as extracted to OCaml.
7. Linear-time computation of the list of key-binding pairs of a finite-map, in sorted order by key, with proofs of its properties.
8. Linear-time combining two finite maps $m_1$ and $m_2$ with function $f$ yielding a map $m$ such that, at any key $i$, $m(i) = f(m_1(i), m_2(i))$; with proofs of its properties.

*Extended Requirements*

9. Efficiency in the face of sparseness: that is, if the magnitude of individual keys is much greater than the number of keys bound in the map.
10. Efficiency not only as extracted to OCaml, but also as represented in Coq and computed within Coq.
11. Extensionality: the property that $(\forall i.\ m_1(i) = m_2(i)) \ \rightarrow\ m_1 = m_2$. This allows Leibniz equality to be used in proofs about the larger systems in which the finite maps are embedded; otherwise, equivalence relations must be used, which is less convenient.
12. Universe polymorphism: a development (program+proof) that uses tries mapping keys to bindings of type $\tau$, and also uses unrelated tries mapping to bindings of type $\sigma$, should not cause unnecessary universe constraints between $\tau$ and $\sigma$.

The *Core Requirements* were all satisfied by a simple binary-trie data structure, implemented in CompCert and used 2005–2021 [9]. In section 2 we will present the data structure and its operations, and discuss the proofs.

In more recent applications, the *Extended Requirements* have been needed. Section 4 describes a new variant of the data structure that has the extensionality property, handles

[1] All proofs mentioned in this paper are machine-checked unless explicitly noted otherwise.

sparseness better, and is significantly more efficient within Coq and somewhat more efficient as extracted to OCaml. Source code for implementations and proofs may be found at `https://github.com/xavierleroy/canonical-binary-tries`.

Extensionality could be achieved in other ways, such as sigma-types in Coq (the dependent pair of a data structure with a proof of its canonicity). We describe alternative implementations in section 6. Our benchmark measurements (section 7) show that our new variant, "canonical binary tries" performs well in all contexts, while each of the alternatives is problematic in at least some contexts.

## 2 PTree: simple binary tries keyed by positive numbers

A simple and efficient data structure for lookup tables (finite maps on integer or string keys) is the *hash table*. But this is not suitable for purely functional implementations, since the update operation is destructive.[2]

Therefore one turns naturally to trees—for example, binary search trees. The lookup operator in a balanced BST takes $\log N$ comparisons, and achieves asymptotic complexity of $\log N$ per operation only if comparison takes constant time. But Coq does not have a primitive integer type with constant-time comparisons.[3]

Coq's logic, the Calculus of Inductive Constructions (CiC) encourages the *construction* of types and operations, instead of *axiomatization*. For example, the natural numbers, and addition upon them, are constructed as,

**Inductive** nat := O : nat | S : nat → nat.

**Fixpoint** add (*n m* : nat) : nat := **match** *n* **with** O ⇒ *m* | S *p* ⇒ S (add *p m*) **end**.

and then properties such as the associativity of add are proved as lemmas instead of being axiomatized.

But this representation of natural numbers is not suitable for use as the keys of efficient lookup tables, because it is essentially a *unary* notation, in which representing the number $n$ takes space proportional to $n$, comparing $n < m$ takes time proportional to $\min(n,m)$, and so on. Therefore users of Coq often use a *binary* representation of integers, constructed as follows:

**Inductive** positive := xI : positive → positive | xO : positive → positive | xH : positive.

**Inductive** Z := Z0 : Z | Zpos : positive → Z | Zneg : positive → Z.

The "meaning" of xH is the positive number 1; xO($p$) represents $2p+0$, and xI($p$) represents $2p+1$. Therefore the number $13_{10} = 1101_2 =$ xI(xO(xI xH)), and the number 0 is unrepresentable—these numbers truly are *positive*, not merely nonnegative.

That means that the Z type (of binary unbounded signed integers) has the property of *extensionality*—no two data structures of type Z represent the same mathematical integer. But the Z type is of no further concern in this paper; we will use positive as the type of keys for lookup tables.

[2] *Persistent arrays* give a functional semantics layered on an imperative (destructive-update) implementation, but these are not suitable because they would require extending the logic's kernel of axioms.

[3] In fact, no programming language has a primitive integer type with constant-time comparisons; they typically have a *range-limited* integer type $-2^k \le i < 2^k$ or a *modular* integer type ($i \bmod 2^k$). Most versions of Coq 2005-present have not had a built-in integer type with fast comparisons; and relying on them would increase the size of the trusted base, that is, Coq kernel code, Coq axioms, Coq extraction to OCaml, and OCaml operations upon which we rely.

The number $p$, represented as a positive, is represented with $\Theta(\log p)$ constructors. When extracted to OCaml, this really means $\log p$ two-word records (in which the first word contains the constructor tag xI/xO/xH and the second word is a pointer to the remainder). That's a lot more than the single word in which a native OCaml program or a C++ program might use to represent an integer key, but it is *efficient enough* in practice, for the CompCert compiler and similar applications.

The positive number $p$, represented within Coq, also has $\log p$ constructors, but each Coq construction is represented within the Coq kernel as two OCaml constructions that *describe* the Coq constructor. So the number $p$ occupies $5 \log p$ words (or $8 \cdot 5 \log p$ bytes on a 64-bit machine). In practice this is usually efficient enough.[4]

Given that Coq has this natural way to represent binary numbers using inductive data types, it is natural to use *binary tries* to implement lookup tables. A *trie* is a directed tree in which every edge is labeled from a finite alphabet; a *word* (such as "cat") is looked up by traversing the path of edges labeled sequentially with the letters of the word (such as "c" then "a" then "t"); and the *nodes* contain values of the range type of the lookup table. A *binary* trie is one in which the alphabet is $\{0,1\}$.

The implementation of binary tries, with keys that are *positive*s, is very straightforward in Coq:

```
Inductive option (A: Type) := Some : A → option A | None : option A.

Inductive tree (A : Type) := Leaf : tree A | Node : tree A → option A → tree A → tree A.

Definition empty {A : Type} : tree A := Leaf.

Fixpoint get {A : Type} (i : positive) (m : tree A) : option A :=
  match m with
  | Leaf ⇒ None
  | Node l o r ⇒ match i with xH ⇒ o | xO i' ⇒ get i' l | xI i' ⇒ get i' r end
  end.

Fixpoint set {A : Type} (i : positive) (v : A) (m : tree A) : tree A :=
 match m with
 | Leaf ⇒ match i with
             | xH ⇒ Node Leaf (Some v) Leaf
             | xO i' ⇒ Node (set i' v Leaf) None Leaf
             | xI i' ⇒ Node Leaf None (set i' v Leaf)
             end
 | Node l o r ⇒ match i with
                  | xH ⇒ Node l (Some v) r
                  | xO i' ⇒ Node (set i' v l) o r
                  | xI i' ⇒ Node l o (set i' v r)
                  end
 end.
```

That is, each node of the tree is either a Leaf, meaning that *no extension of the key leading to this node is in the domain of the map*, or an internal Node with left subtree $l$, optional binding-at-this-node $o$, and right subtree $r$. If $o$=None then the key leading to this node is not in the domain of the map, but some extensions of it might be. If $o$=Some $v$ then the key leading to this node is mapped to $v$, and extensions of the key might also be mapped (in $l$

[4] In OCaml, an arity-$k$ constructor applied to arguments takes $k+1$ words, including one for the constructor-tag. In Coq, an arity-$k$ constructor is represented as one arity-2 OCaml construction plus one length-$k$ array, which take (respectively) 3 words and $k+1$ words as represented in OCaml within the Coq kernel. The arity-2 OCaml construction also points to a description of the constructor, but that description is shared among all its uses so we don't count it.

and $r$). The edge from this node to subtree $l$ is implicitly labeled by 0 (or xO), and the edge to subtree $r$ is implicitly labeled by 1 (or xI).

Looking up a key $p$ in a trie takes time proportional to the number of constructors representing the key, which is about $\log_2 p$; update is similarly efficient; this is *efficient enough* for many important applications, including CompCert.

From these definitions it is easy to prove the important algebraic properties of maps:

```
Theorem gempty: (* "get-empty" *)
   ∀(A: Type) (i: positive), get i (empty A) = None.
Proof. induction i; simpl; auto. Qed.

Theorem gss: (* "get-set-same" *)
  ∀(A: Type) (i: positive) (x: A) (m: tree A), get i (set i x m) = Some x.
Proof. induction i; destruct m; simpl; auto. Qed.

Lemma gleaf : ∀(A : Type) (i : positive), get i (Leaf : tree A) = None.
Proof. exact gempty. Qed.

Theorem gso: (* "get-set-other" *)
   ∀(A: Type) (i j: positive) (x: A) (m: tree A), i <> j → get i (set j x m) = get i m.
Proof. induction i; intros; destruct j; destruct m; simpl;
        try rewrite <- (gleaf A i); auto; try apply IHi; congruence.
Qed.
```

As usual in machine-checked proof scripts, the reader is not necessarily expected to read and understand the proofs; but the *theorem statements* should be clear, and `induction i; destruct m` suggests that the proof is by induction on the positive number $i$ and then case analysis on $m$.

In less than 50 lines of Coq we have accomplished the first 6 of the core requirements: the operators `empty`, `get`, `set`, proofs of the algebraic laws, a purely functional implementation with a persistent semantics, an efficient-enough representation of keys, and efficient-enough asymptotic time complexity for many programs extracted to OCaml.

To produce an association list of the key-binding pairs, one *could* write a function such as,

```
Fixpoint elements {A} (m: tree A) (i: positive) : list (positive * A) :=
 match m with
 | Leaf ⇒ nil
 | Node l o r ⇒
    elements l (xO i)
   ++ match o with None ⇒ nil | Some x ⇒ (prev i, x)::nil end
   ++ elements r (xI i)
   end.
```

where `prev` is positive-reversal, so `prev(xI (xI (xO (xH))) = xO (xI (xI xH))`. However, this implementation is inefficient: since list-concatentation ++ takes time proportional to the length of its first argument, this function will take about $N \log N$ time for a tree containing $N$ elements. So instead, one implements `elements` by accumulating the list to the right of the current tree, costing time exactly linear in the number of nodes in the tree. We will not show the details.

Proofs of the properties of the `elements` function are reasonably straightforward, though more complex than the `gso` theorem.

It is easy to define collective operations such as "map" and "filter" that transform a finite map into another finite map. For example, here is a `map_filter` $f$ operation that applies a function $f$: $A \to$ `option` $B$ to every value $a$ contained in a `tree` $A$ map, discarding the value if $f\ a =$ `None`.

```
Fixpoint map_filter {A B} (f: A → option B) (m: tree A) : tree B :=
  match m with
  | Leaf ⇒ Leaf
  | Node l None r ⇒ Node' (map_filter f l) None (map_filter f r)
  | Node l (Some a) r ⇒ Node' (map_filter f l) (f a) (map_filter f r)
  end.
```

In the second and third cases, if the recursive calls map_filter f l and map_filter f r both return Leaf and f a returns None, we do not want to build a useless empty subtree Node Leaf None Leaf. Empty nonleaf subtrees are not *wrong,* but they waste memory. So the solution is to use a Node' pseudoconstructor function, as follows:

```
Definition Node' {A: Type} (l: tree A) (x: option A) (r: tree A): tree A :=
match l, x, r with Leaf, None, Leaf ⇒ Leaf | _, _, _ ⇒ Node l x r end.
```

Many applications wish to combine two finite maps $m_1$ : tree $A$, $m_2$ : tree $B$ with a user-supplied function $f$: option $A \to$ option $B \to$ option $C$. We assume that $f$ None None = None, and we require that combine $m_1$ $m_2 = m$ if and only if $\forall i, f(m_1(i))(m_2(i)) = m(i)$.

One might propose an implementation something like this:

```
Fixpoint combine (m1: tree A) (m2: tree B) : tree C :=
 match m1, m2 with
 | Leaf, Leaf ⇒ Leaf
 | Leaf, Node l2 o2 r2 ⇒ Node' (combine Leaf l2) (f None o2) (combine Leaf r2)
 | Node l1 o1 r1, Leaf ⇒ Node' (combine l1 Leaf) (f o1 None) (combine r1 Leaf)
 | Node l1 o1 r1, Node' l2 o2 r2 ⇒ Node (combine l1 l2) (f o1 o2) (combine r1 r2)
 end.
```

but there is a problem here. Some of the recursive calls use substructures of $m_1$ while others use substructures of $m_2$, so this is not actually legal in Coq. The solution is that once a Leaf is reached on either side, we can traverse the other side using map_filter, which is recursive on just one tree.

```
Fixpoint combine (m1: tree A) (m2: tree B) : tree C :=
  match m1, m2 with
  | Leaf, _ ⇒ map_filter (fun b ⇒ f None (Some b)) m2
  | _, Leaf ⇒ map_filter (fun a ⇒ f (Some a) None) m1
  | Node l1 o1 r1, Node l2 o2 r2 ⇒ Node' (combine l1 l2) (f o1 o2) (combine r1 r2)
end.
```

This combine function is quite general and reasonably efficient. It takes time linear in the total sizes of the input trees. Proofs of its properties are straightforward, for example,

```
Theorem gcombine: ∀(m1: tree A) (m2: tree B) (i: positive),
      get i (combine m1 m2) = f (get i m1) (get i m2).
```

The proof is only 6 lines of Coq.

*Summary of Section 2.* Binary tries, with keys that are constructive binary positive numbers, are a simple data structure with an easy implementation and reasonably straightforward proofs. They are efficient enough in practice for many useful applications, especially as extracted into OCaml.

## 3 The need for extensionality

*Extensionality* is a property of equality in formal logics. It says that two objects are equal if they have the same external properties. For example, the extensionality property for functions says that two functions $f$ and $g$ are equal as soon as they have the same domain and satisfy $\forall x,\ f(x) = g(x)$. This extensionality property holds in set theory but not in Coq's type theory.

For finite maps, the extensionality property says that two maps are equal if they are equivalent, that is, if they map equal keys to equal values. Like most popular implementations of finite maps, binary tries (represented as in Section 2) are not extensional. That is, equivalent tries are not necessarily equal, as shown by the following example.

$$\begin{array}{lll} m_1 & \cdot\ 1\ ((\cdot\ 2\ \cdot) \circ (\cdot\ 2\ \cdot)) & \cdot \text{ represents } \mathsf{Leaf} \\ m_2 & (\cdot \circ \cdot)\ 1\ ((\cdot\ 2\ \cdot) \circ (\cdot\ 2\ \cdot)) & \circ \text{ represents } \mathsf{None} \end{array}$$

Here, $m_1$ and $m_2$ are equivalent (for any $i$, $\mathsf{get}\ i\ m_1 = \mathsf{get}\ i\ m_2$), but $m_2$ has an extra empty left subtree $\mathsf{Node\ Leaf\ None\ Leaf}$ where $m_1$ has just $\mathsf{Leaf}$.

This is not an insurmountable problem. One can reason using equivalence rather than equality. Coq has support for such "Setoid" reasoning; but it's still inconvenient, and a truly canonical representation of tries would simplify many definitions and proofs. For example, consider the following two algebraic laws of finite maps:

$$\begin{aligned} \mathsf{set}\ k\ v\ m &= m \qquad \text{if } \mathsf{get}\ k\ m = \mathsf{Some}\ v \\ \mathsf{set}\ k_1\ v_1\ (\mathsf{set}\ k_2\ v_2\ m) &= \mathsf{set}\ k_2\ v_2\ (\mathsf{set}\ k_1\ v_1\ m) \end{aligned}$$

If the representation of $m$ were canonical (which the binary tries of Section 2 are not), then these laws would be trivial consequences of extensionality. In the absence of extensionality, these laws require specific proofs, using the definition of set and inductions over $m$.

More generally, a large Coq development might have types that contain finite maps, such as a type for "program modules" in which finite maps represent bindings of names to definitions. If the finite maps do not support Leibnitz equality, and one must therefore use setoids, then equality reasoning about "program modules" must also use setoids.

Coq proofs with setoids are larger and slower than proofs with Leibnitz equality. Extensional tries (permitting Leibnitz equality) could perform much faster in Coq proofs. And indeed they do; see section 8.

One might think, "extra empty subtrees will never arise in practice, since the set operation cannot produce them." But then one would have to maintain the invariant that a given trie was produced by a sequence of set operations.

One might think, "we'll maintain that invariant using dependent types containing proofs that the tree has no empty subtrees." Section 6 describes two attempts along this line and the efficiency problems they cause.

Therefore, what we want is a *first-order, naturally extensional representation* of tries.

## 4 Canonical binary tries

For a first-order naturally extensional representation that handles sparseness better, our solution is to make an inductive datatype that *cannot* represent an empty mapping. The data structure is as follows:

```
Inductive tree' (A: Type) : Type :=
 | Node001: tree' A → tree' A                          (* only a right subtree *)
 | Node010: A → tree' A                                (* only a middle value *)
 | Node011: A → tree' A → tree' A                      (* only middle and right *)
 | Node100: tree' A → tree' A                          (* only a left subtree *)
 | Node101: tree' A → tree' A → tree' A                (* left, right, no middle *)
 | Node110: tree' A → A → tree' A                      (* only left and middle *)
 | Node111: tree' A → A → tree' A → tree' A.           (* left, middle, right *)

Inductive tree (A: Type) : Type :=
  | Empty : tree A
  | Nodes: tree' A → tree A.
```

Each of the three digits in the name of a node constructor represents the presence or absence of the left, middle, or right components of the node. Thus, Node101 $l$ $r$ represents a node with left and right subtrees but no middle, and Node011 $x$ $r$ represents what would have been written (in our previous representation) as Node Leaf (Some $x$) $r$.

A crucial property of this data structure is *canonicity*: for any finite set $S$ of (key, value) bindings, there is only one value of type tree that represents this set. If $S$ is empty, it is represented by Empty, and cannot be represented by Nodes $t$, as a value $t$ of type tree' always contains at least one (key, value) binding. If $S$ is not empty, it can only be represented by Nodes $t$. The head constructor of $t$ is uniquely determined by whether the sets

$$S_h = \{(\mathsf{xH}, v) \mid (\mathsf{xH}, v) \in S\}$$
$$S_o = \{(i, v) \mid (\mathsf{xO}\ i, v) \in S\}$$
$$S_i = \{(i, v) \mid (\mathsf{xI}\ i, v) \in S\}$$

are empty or not. Induction on the size of $S$ shows that the subtrees of $t$, if any, are uniquely determined.

Canonicity implies extensionality, of course: two values of type tree that represent the same set of (key, value) pairs are equal. In addition, canonicity improves computational efficiency compared with the noncanonical binary tries from section 2: no memory is consumed to represent empty left, middle, or right components of a node. As the experimental evaluation in sections 7 and 8 shows, the gains in memory usage and execution times are significant, especially for sparse maps.

Defining the lookup and update functions is not quite as simple as before:

```
Definition empty (A : Type) := (Empty : tree A).

Fixpoint get' {A} (p: positive) (m: tree' A) : option A :=
  match p, m with
  | xH, Node001 _ ⇒ None
  | xH, Node010 x ⇒ Some x
  | xH, Node011 x _ ⇒ Some x
  | xH, Node100 _ ⇒ None
  | xH, Node101 _ _ ⇒ None
  | xH, Node110 _ x ⇒ Some x
  | xH, Node111 _ x _ ⇒ Some x
  | xO q, Node001 _ ⇒ None
  | xO q, Node010 _ ⇒ None
  | xO q, Node011 _ _ ⇒ None
  | xO q, Node100 m' ⇒ get' q m'
  | xO q, Node101 m' _ ⇒ get' q m'
  | xO q, Node110 m' _ ⇒ get' q m'
  | xO q, Node111 m' _ _ ⇒ get' q m'
  | xI q, Node001 m' ⇒ get' q m'
```

```
  | xI q, Node010 _ ⇒ None
  | xI q, Node011 _ m' ⇒ get' q m'
  | xI q, Node100 m' ⇒ None
  | xI q, Node101 _ m' ⇒ get' q m'
  | xI q, Node110 _ _ ⇒ None
  | xI q, Node111 _ _ m' ⇒ get' q m'
end.

Definition get {A} (p: positive) (m: tree A) : option A :=
  match m with Empty ⇒ None | Nodes m' ⇒ get' p m' end.
```

The get' function has 21 cases, handling each combination of one of the 3 positive constructors and one of the 7 node constructors.

The update function set follows the same pattern. An update to the empty tree is handled by the set0 function, which creates a nonempty tree with a single branch:

```
Fixpoint set0 {A} (p: positive) (x: A) : tree' A :=
match p with
| xH ⇒ Node010 x
| xO q ⇒ Node100 (set0 q x)
| xI q ⇒ Node001 (set0 q x)
end.
```

To update a nonempty tree, the function set' performs the same 21-case analysis as get':

```
Fixpoint set' {A} (p: positive) (x: A) (m: tree' A) : tree' A :=
match p, m with
| xH, Node001 r ⇒ Node011 x r
| xH, Node010 _ ⇒ Node010 x
| xH, Node011 _ r ⇒ Node011 x r
| xH, Node100 l ⇒ Node110 l x
| xH, Node101 l r ⇒ Node111 l x r
| xH, Node110 l _ ⇒ Node110 l x
| xH, Node111 l _ r ⇒ Node111 l x r
| xO q, Node001 r ⇒ Node101 (set0 q x) r
| xO q, Node010 y ⇒ Node110 (set0 q x) y
| xO q, Node011 y r ⇒ Node111 (set0 q x) y r
| xO q, Node100 l ⇒ Node100 (set' q x l)
| xO q, Node101 l r ⇒ Node101 (set' q x l) r
| xO q, Node110 l y ⇒ Node110 (set' q x l) y
| xO q, Node111 l y r ⇒ Node111 (set' q x l) y r
| xI q, Node001 r ⇒ Node001 (set' q x r)
| xI q, Node010 y ⇒ Node011 y (set0 q x)
| xI q, Node011 y r ⇒ Node011 y (set' q x r)
| xI q, Node100 l ⇒ Node101 l (set0 q x)
| xI q, Node101 l r ⇒ Node101 l (set' q x r)
| xI q, Node110 l y ⇒ Node111 l y (set0 q x)
| xI q, Node111 l y r ⇒ Node111 l y (set' q x r)
end.

Definition set {A} (p: positive) (x: A) (m: tree A) : tree A :=
match m with
| Empty ⇒ Nodes (set0 p x)
| Nodes m' ⇒ Nodes (set' p x m')
end.
```

Proofs of the fundamental theorems (such as gss and gso) are still short and easy: about 25 lines in all, including supporting lemmas.

Other functions that operate over a single tree remain reasonably easy to write. For example, here is the recursive part of the map_filter function that applies a function $f$: $A \to$ option $B$ to a nonempty tree $m$: tree' $A$:

```
Fixpoint map_filter' {A B} (f: A → option B) (m: tree' A) : tree B :=
  match m with
  | Node001 r ⇒ Node Empty None (map_filter' f r)
  | Node010 x ⇒ Node Empty (f x) Empty
  | Node011 x r ⇒ Node Empty (f x) (map_filter' f r)
  | Node100 l ⇒ Node (map_filter' f l) None Empty
  | Node101 l r ⇒ Node (map_filter' f l) None (map_filter' f r)
  | Node110 l x ⇒ Node (map_filter' f l) (f x) Empty
  | Node111 l x r ⇒ Node (map_filter' f l) (f x) (map_filter' f r)
  end.
```

The Node function used in the definition above is similar to the Node constructor and the Node' smart constructor of section 2: given a possibly empty left subtree, a possibly absent value, and a possibly empty right subtree, it returns the corresponding tree.

```
Definition Node {A} (l: tree A) (o: option A) (r: tree A) : tree A :=
  match l,o,r with
  | Empty, None, Empty ⇒ Empty
  | ... 2 cases elided
  | Empty, Some x, Nodes r' ⇒ Nodes (Node011 x r')
  | ... 4 cases elided
  end.
```

The case analysis performed by Node can be partially redundant. For instance, in the first case of the map_filter' function, we use Node with l = Empty and o = None; we would prefer to avoid discriminating over l and o in Node. This is easily achieved by asking Coq to unfold the definition of Node inside map_filter', then simplify using call-by-value evaluation:

```
Definition fast_map_filter' := Eval cbv [Node] in @map_filter'.
```

The resulting function performs the minimal number of case distinctions on the results of f and of recursive calls. Hence, it executes faster than map_filter', both within Coq and after extraction to OCaml. At the same time, fast_map_filter' is *convertible* with map_filter', since it is obtained by reductions, hence the two functions are definitionally equal, and all the results proved on the nicer-looking map_filter' hold for the faster fast_map_filter'.

Functions that operate over two trees, such as the combine function from Section 2, are more difficult to write because of the high number of cases. Consider the recursive case of the combine function, the one that operates on two nonempty trees in parallel.

```
Variables A B C: Type.
Variable f: option A → option B → option C.
Definition combine'_l := map_filter' (fun a ⇒ f (Some a) None).
Definition combine'_r := map_filter' (fun b ⇒ f None (Some b)).

Fixpoint combine' (m1: tree' A) (m2: tree B) {struct m1} : tree C :=
  match m1, m2 with
  | ...
  | Node101 l1 r1, Node011 x2 r2 ⇒ Node (combine'_l l1) (f None (Some x2)) (combine' r1 r2)
  | ...
  end.
```

A naive case analysis on m1 and m2 results in 49 cases. Each case is simple enough: we call combine' if there are two left subtrees or two right subtrees, combine'_l if there is only a left subtree, combine'_r if there is only a right subtree. Likewise, reasoning about these 49 cases is relatively easy, using Coq's tactic language for automation. But writing all the cases in the first place is a pain!

Even so, *proofs* about the 49-case combine' function can be concise. Consider this theorem, about the interaction of combine' with get':

```
Lemma gcombine': ∀m1 m2 i,
      get i (combine' m1 m2) = f (get' i m1) (get' i m2).
Proof.
   induction m1; destruct m2; intros; simpl; rewrite gNode;
   destruct i; simpl; auto using gcombine'_l, gcombine'_r.
Qed.
```

Coq's tactic language allows the chaining of induction on m1, case analysis (destruct) on m2, simplification, rewriting, and case analysis on i, all in two lines of Ltac scripting.

## 5 Defining functions with case analysis, in Coq

Our 49-case combine' function has a regular structure that is motivated by the way that canonical tries relate to original tries. We briefly digress to show how to define such functions—that do a large (but regularly structured) case analysis—concisely in Coq. We show two approaches: tactical function definition, and *views*.

### 5.1 Tactical function definition.

Here is a definition combine'_by_tac that produces the same function as the 49-case combine' described above:

```
Fixpoint combine'_by_tac (m1: tree' A) (m2: tree' B) {struct m1} : tree C.
Proof.
  destruct m1 as [ r1 | x1 | x1 r1 | l1 | l1 r1 | l1 x1 | l1 x1 r1 ];
  destruct m2 as [ r2 | x2 | x2 r2 | l2 | l2 r2 | l2 x2 | l2 x2 r2 ];
  (apply Node;
   [ solve [ exact (combine'_by_tac l1 l2)
           | exact (combine'_l l1)
           | exact (combine'_r l2)
           | exact Empty]
   | solve [ exact (f (Some x1) (Some x2))
           | exact (f None (Some x2))
           | exact (f (Some x1) None)
           | exact None]
   | solve [ exact (combine'_by_tac r1 r2)
           | exact (combine'_l r1)
           | exact (combine'_r r2)
           | exact Empty ]
   ]).
Defined.
```

The first line describes a *goal:* define a recursive function of a particular type by structural induction on m1. The "Proof" is a case analysis describe in Coq's tactic language: the destruct builds a pattern-**match** into the "proof" (in this case, into the "program"). So we start with two nested 7-case matches, that is, a 49-case double pattern match. Each one of those has a right-hand-side that's an application of the Node pseudo constructor, so we say apply Node in the tactic language. Since Node is a 3-argument function, we have three subgoals remaining, separated by the punctuation [ | | ]. In each case, we try four different things (the exact tactics), but in each case, only one of those four will type-check, and that is the one that will be chosen for use in defining the function.

The reader may not be confident that this defines the right function! Indeed, the author of the tactic may not be confident. One can gain confidence by this theorem, which shows that the new implementation is $\beta\delta$-convertible with the hand-written 49-case version:

```
Lemma combine'_by_tac_eq: combine'_by_tac = combine'.
Proof. reflexivity. Qed.
```

However, the purpose of this tactic is to *avoid* having to write the 49-case version by hand. So a more useful theorem to prove correctness of the tactical implementation (or any implementation) is that it combine' interacts properly with get'. And indeed, the same gcombine' lemma-statement (as shown in §4) *and exactly the same tactical proof* works to prove that theorem for combine'_by_tac as for the direct implementation of combine'.

## 5.2 Views

Instead of definition-by-tactic, another way to avoid explosion in the number of cases is to work with canonical binary tries using a *view* consisting of two cases only: either an empty leaf or a node consisting of a left subtree, an optional value, and a right subtree. In other words, the view is isomorphic to the concrete representation of the original binary tries from section 2. One half of this view—the one that lets us construct values of type tree A—consists of the Empty constructor and the Node function mentioned above:

```
Empty : ∀A, tree A
Node : ∀A, tree A → option A → tree A → option A
```

The other half of the view—the one that lets us inspect and recurse over values of type tree A—is provided by elimination and induction principles. Here is a simple, nonrecursive case analysis with two cases:

```
Definition tree_case {A B} (empty: B)
                           (node: tree A → option A → tree A → B)
                           (m: tree A) : B :=
  match m with
    | Empty ⇒ empty
    | Nodes (Node001 r) ⇒ node Empty None (Nodes r)
    | Nodes (Node010 x) ⇒ node Empty (Some x) Empty
    | Nodes (Node011 x r) ⇒ node Empty (Some x) (Nodes r)
    | Nodes (Node100 l) ⇒ node (Nodes l) None Empty
    | Nodes (Node101 l r) ⇒ node (Nodes l) None (Nodes r)
    | Nodes (Node110 l x) ⇒ node (Nodes l) (Some x) Empty
    | Nodes (Node111 l x r) ⇒ node (Nodes l) (Some x) (Nodes r)
    end.
```

We expect this case analysis principle to satisfy the two equations

```
tree_case empty node Empty = empty
tree_case empty node (Node l o r) = node l o r
```

The second equation is not true in general: in the case where l and r are Empty and o is None, Node l o r is Empty and the tree_case analysis returns empty, whereas our second equation requires it to return node Empty None Empty. To support the equation, we can require that node Empty None Empty = empty, showing that the two cases of the analysis agree in this special case. Alternatively, we can rule out the special case by adding the precondition not_trivially_empty l o r to the second equation. The not_trivially_empty predicate is defined as

```
Definition not_trivially_empty {A} (l: tree A) (o: option A) (r: tree A) :=
  match l, o, r with
  | Empty, None, Empty ⇒ False
  | _, _, _ ⇒ True
  end.
```

The custom induction principle that we define later in this section provides us with the not_trivially_empty l o r guarantee in the Node l o r case. Hence we can always apply the tree_case equations when reasoning with this induction principle.

A minor variant of tree_case implements a recursion principle. We still have two cases, empty and node, but the node case also receives the results of recursing over the left and right subtrees.

```
tree_rec: ∀{A B} (empty: B)
                 (node: tree A → B → option A → tree A → B → B),
                 tree A → B.
```

Along the same lines, we can also define a general, dependently typed induction principle, usable both for computations and for proofs. Its type is similar to that of the Coq-generated induction principle for the two-constructor tree type of section 2, except that the node case also receives a proof of the not_trivially_empty guarantee:

```
tree_ind: ∀{A: Type} (B: A → Type)
                 (empty: B Empty)
                 (node: ∀l, B l → ∀o r, B r → not_trivially_empty l o r → B (Node l o r)),
                 (m: tree A), B m.
```

One use for this induction principle is to simplify proofs about hand-written functions over type tree. For example, to prove properties of the set function defined above, we can first show the following equations:

```
      set xH v Empty = Node Empty (Some v) Empty
  set (xO q) v Empty = Node (set q v Empty) None Empty
   set (xI q) v Empty = Node Empty None (set q v Empty)
 set xH v (Node l o r) = Node l (Some v) r
set (xO q) v (Node l o r) = Node (set q v l) o r
 set (xI q) v (Node l o r) = Node l o (set q v r)
```

which follow by a trivial case analysis on l, o and r. Then, we can prove properties such as get $i$ (set $i$ $v$ $m$) = Some $v$ by an outer structural induction over $i$ and an inner induction over $m$ that uses the tree_ind custom induction principle. This reduces the number of cases to consider to $3 \times 2 = 6$ cases.

Custom case analysis and induction principles can also be used to define functions over trees. For instance, the infamous combine function can be defined quite concisely by:

```
tree_rec combine_r
         (fun l1 rec_l1 o1 r1 rec_r1 m2 ⇒
           tree_case
             (combine_l (Node l1 o1 r1))
             (fun l2 o2 r2 ⇒ Node (rec_l1 l2) (f o1 o2) (rec_r1 r2))
             m2)
```

This definition avoids the risk of carpal tunnel syndrom associated with the hand-written, 49-case definition. However, the tree_rec and tree_case functions add significant run-time overhead. We can recover most of the efficiency of the hand-written version by partial evaluation within Coq: as shown above for map_filter', we can use Eval cbv to force Coq to expand the tree_rec and tree_case functions, then simplify. Even then, a minor inefficiency remains: the second argument to combine is tested for emptiness at each recursive step over the first argument, instead of being tested once and for all in the hand-written or tactic-based implementations. A custom double induction principle, operating over two trees at once, can remove this last inefficiency.

To conclude this discussion: the canonical representation of binary tries greatly increases the number of cases in function definitions and in proofs. This increase can be challenging

for functions operating on two or more tries at the same time, but can be managed either by writing tactics that define these functions, or by defining appropriate case analysis and induction principles that reduce the number of cases, then using Coq's built-in evaluation facilities to remove the execution overhead caused by these principles.

## 6 Other approaches to extensionality

The canonical, first-order representation we have just presented in section 4 is not the only way to achieve extensionality: another way is to use dependent types, as we now demonstrate. The reader may choose to skip this section, because it turns out that none of these other approaches performs as well, in a wide enough range of applications, as the canonical representation.

### 6.1 Subset types

The simple tries from section 2 are not extensional because they might contain subtrees Node Leaf None Leaf, which contain no elements but structurally differ from Leaf. The extensionality property holds if we restrict ourselves to trees that satisfy the following invariant wf, "well-formed: the tree contains no empty nodes Node Leaf None Leaf."

```
Definition not_trivially_empty {A} (l: tree A) (o: option A) (r: tree A) :=
  match l, o, r with
  | Leaf, None, Leaf ⇒ False
  | _, _, _ ⇒ True
  end.

Inductive wf {A}: tree A → Prop :=
  | wf_Leaf: wf Leaf
  | wf_Node: ∀l o r, wf l → wf r → not_trivially_empty l o r → wf (Node l o r).
```

It is not difficult to show that if wf $m_1$ and wf $m_2$ hold, and if $m_1$ and $m_2$ are equivalent (for any $i$, get $i$ $m_1$ = get $i$ $m_2$), then $m_1$ and $m_2$ are structurally equal.

To maintain the invariant through all operations over trees, we define the type of tries $m$ that satisfy wf $m$ as a Coq subset type, also called sigma-type:

```
Definition t (A: Type) : Type := { m: tree A | wf m }.
```

Values of this type are dependent pairs of a tree $m$ and a proof of wf $m$. This approach is used in the Std++ library [5]. Earlier, it was used by Filliâtre and Letouzey in their implementations of finite sets and finite maps by AVL trees [6], to maintain the invariant that all trees considered are binary search trees.

A strength of this approach is that it reuses the previous implementation of tries (definitions of operations and proofs of properties), simply wrapping them up with proofs of the wf invariant.

```
Fixpoint set_raw {A : Type} (i : positive) (v : A) (m : tree A) : tree A :=
  match m with ... end.

Lemma set_wf: ∀{A: Type} i (v: A) {m: tree A}, wf m → wf (set_raw i v m).
Proof. ... Qed.

Definition set {A: Type} (i: positive) (v: A) (m: t A) : t A :=
  exist _ (set_raw i v (proj1_sig m)) (set_wf i v (proj2_sig m)).
```

The "subset type" approach executes efficiently after extraction: propositions and proofs are erased during extraction, hence the type t is identical to tree in the extracted OCaml code, and there is no overhead compared with the original implementation of tries.

However, execution within Coq is inefficient because the proof of wf $m$ grows linearly *in the number of operations performed over m*. In effect, the proof part of a value of type t A contains the history of all operations applied to produce this value.

Consider for instance the result of setting the key xH to values $v_1, \ldots, v_n$ successively. The value part is the small tree Node Leaf (Some $v_n$) Leaf, but the proof part is set_wf xH $v_n$ (… (set_wf xH $v_1$ wf_Leaf) …), a term of size $O(n)$. Assuming the proof of lemma set_wf is opaque, this proof term does not reduce and takes $O(n)$ space and $O(n)$ time when normalization is requested.

We experimented with alternate, "transparent" definitions of set_wf and related lemmas that would reduce to small proof terms during normalization, but failed to find definitions that would be efficient both in space and in normalization time.

### 6.2 A poor man's inductive-inductive definition

Rather than maintaining the wf invariant as a separate proof term, as in section 6.1, we could try to make it part of the main tree data structure, along the following lines:

```
Inductive tree (A : Type) :=
  | Leaf : tree A
  | Node : ∀(l: tree A) (o: option A) (r: tree A),   not_trivially_empty l o r → tree A.
```

The Node constructor carries not only an optional value and left and right subtrees, but also a proof that the node is not trivially empty. We would expect this representation to compute within Coq more efficiently than the subset type representation, since proofs of not_trivially_empty are small, and only those proofs relevant to the nodes of the tree are kept.

The definition above is incorrect, of course, since the not_trivially_empty predicate mentions the Leaf constructor of the type tree that we are defining. We need to define the predicate and the tree simultaneously, along the following lines:

```
Inductive tree (A : Type) :=
  | Leaf : tree A
  | Node : ∀(l: tree A) (o: option A) (r: tree A),   not_trivially_empty l o r → tree A.

with not_trivially_empty {A: Type}: tree A → option A → tree A → Prop :=
  | not_trivially_empty_intro: ∀l o r,
          ∼(l = Leaf ∧ o = None ∧ r = Leaf) → not_trivially_empty l o r.
```

This is called an *inductive-inductive definition*: the type tree is defined mutually recursively with the type family not_trivially_empty that is indexed over type tree.

Such inductive-inductive types are supported in Agda but not in Coq. However, our use of induction-induction is so simple that we can work around this limitation of Coq. We index the type of trees with an additional parameter of type kind, with the kind Empty standing for the constructor Leaf, and the kind Nonempty standing for the constructor Node:

```
Inductive kind : Type := Empty | Nonempty.

Definition not_trivially_empty {A: Type} (kl: kind) (o: option A) (kr: kind) :=
  ∼(kl = Empty ∧ o = None ∧ kr = Empty).

Inductive tree (A : Type) : kind → Type :=
```

```
| Leaf : tree A Empty
| Node : ∀(kl kr: kind) (l: tree A kl) (o: option A) (r: tree A kr),
                not_trivially_empty kl o kr → tree A Nonempty.
```

This breaks the circularity in the definition of not_trivially_empty and turns tree into a regular inductive type. Finally, a well-formed tree is defined as a dependent pair of a kind and a tree of that kind:

```
Inductive t (A: Type) : Type :=
  | Tree : ∀(k: kind), tree A k → t A.
```

With some elbow grease, we can adapt the definitions of operations over trees, first at type tree A k, then at type t A after abstracting over the kind k. For example, the set operation looks as follows:

```
Fixpoint set' {A: Type} (k: kind) (i: positive) (v: A) (m: tree A k) : tree A Nonempty := ...

Definition set {A: Type} (i: positive) (v: A) (m: t A) : t A :=
  let '(Tree k m) := m in Tree Nonempty (set' k i v m).
```

Concerning efficiency of evaluation, canonical binary tries seem to have the same asymptotic time and space complexity as the original, nonextensional implementation from section 2, both after extraction and within Coq—see Table 1. However, the Node constructor is bigger by a constant factor: in Coq, it additionally carries two kinds (kl and kr) and a proof of not_trivially_empty; after extraction to OCaml, it still carries the two kinds kl and kr, even if they do not participate in the computations. Consequently, the constant factors for space and time complexity are noticeably higher than in the original implementation.

## 7 Benchmarks

### 7.1 Performance of binary trie implementations

We now evaluate the performance of the various binary trie implementations mentioned in this paper, on synthetic benchmarks, both for execution within Coq or after extraction to OCaml. Here are the implementations:

**Original**: The original, nonextensional implementation described in section 2:

```
Inductive tree (A : Type) := Leaf : tree A | Node : tree A → option A → tree A → tree A.
```

**Canonical**: The canonical binary tries of section 4, based on space-efficient representation:

```
Inductive tree' (A: Type) : Type :=
  | Node001: tree' A → tree' A
  | Node010: A → tree' A
  | Node011: A → tree' A → tree' A
  | Node100: tree' A → tree' A
  | Node101: tree' A → tree' A → tree' A
  | Node110: tree' A → A → tree' A
  | Node111: tree' A → A → tree' A → tree' A.
Inductive tree (A: Type) : Type :=
  | Empty : tree A
  | Nodes: tree' A → tree A.
```

**Node01**: A nonextensional implementation that avoids storing option values in nodes, using two constructors Node0 (node carrying no value) and Node1 (node carrying a value) instead:

```
Inductive tree (A : Type) :=
  | Leaf : tree A
  | Node0 : tree A → tree A → tree A.
  | Node1 : tree A → A → tree A → tree A.
```

This is a middle point between the Original and the Canonical implementations, both in terms of space efficiency and in terms of complexity of implementation.

**Sigma**: An extensional implementation built on top of the Original implementation using a subset type to guarantee well-formedness of trees, as in section 6.1:

```
Inductive tree (A : Type) := Leaf : tree A | Node : tree A → option A → tree A → tree A.
Definition t (A: Type) : Type := { m: tree A | wf m }.
```

**GADT**: An extensional implementation based on an indexed type, also known as a GADT (generalized algebraic data type), as described in section 6.2:

```
Inductive tree (A : Type) : kind → Type :=
  | Leaf : tree A Empty
  | Node : ∀(kl kr: kind) (l: tree A kl) (o: option A) (r: tree A kr),
                not_trivially_empty kl o kr → tree A Nonempty.
Inductive t (A: Type) : Type :=
  | Tree : ∀(k: kind), tree A k → t A.
```

**AVL**: The implementation of finite maps by Filliâtre and Letouzey [6], using AVL balanced binary search trees. It is much more versatile than our binary tries, as it supports keys of any type equipped with a decidable total ordering, not just positive numbers. We include this implementation in the comparison because it is the most efficient finite map data structure provided by the Coq standard library.

We compare the performance of these implementations using three benchmarks:

**Dense**: The keys 1 to 2048 are successively inserted in a trie, then every key is looked up. This produces a dense, perfectly balanced binary trie. This is representative of how binary tries are used in the back-end of the CompCert compiler to represent control-flow graphs, in particular.

**Sparse**: As in the Dense test, we insert then look up a number of keys. The keys correspond to words randomly chosen from an English dictionary, then converted to positive numbers by viewing the words as strings of bits, with 8 bits per character and an ASCII encoding. We have 5064 words with length ranging from 1 to 18 characters (average 8 characters). This test produces sparse, poorly balanced tries with long, nearly empty branches. It is representative of some uses of binary tries in the VST infrastructure, where keys are encodings of program identifiers. The static analyses of the CompCert back-end also use sparse tries intensively.

**Repeated**: We insert the same seven keys (1 to 7) a million times in a trie. The purpose of this test is to check that the trie remains small (7 nodes) and does not grow at each insertion. It is representative of the use of binary tries as environments in a reference interpreter: there are few variables in the program being interpreted, but they are assigned many times.

For execution within Coq, we use `vm_compute` as our primary evaluation mechanism. It implements call-by-value through compilation to a virtual machine. For comparison, we also give some numbers obtained with the `cbv` and `cbn` mechanisms. These are interpreters that follow either call-by-value (`cbv`) or call-by-name plus some sharing of computations (`cbn`). These mechanisms are much slower than `vm_compute`, hence we report results for the Sparse benchmark only. We also extract the benchmarks to OCaml and compile them to

| *Implementation* | Original | Node01 | Canonical | Sigma | GADT | AVL |
|---|---|---|---|---|---|---|
| **Coq execution, `vm_compute`, Sparse test** | | | | | | |
| *Time in s* | 1.92E-02 | 1.87E-02 | 1.51E-02 | 2.00E-02 | 2.35E-02 | 2.26E-01 |
| *Relative time* | 100% | 98% | 79% | 104% | 122% | 1176% |
| **Coq execution, `vm_compute`, Dense test** | | | | | | |
| *Time in s* | 1.16E-03 | 1.14E-03 | 1.22E-03 | 1.45E-03 | 1.55E-03 | 3.38E-02 |
| *Relative time* | 100% | 98% | 106% | 125% | 134% | 2915% |
| **Coq execution, `vm_compute`, Repeated test** | | | | | | |
| *Time in s* | 5.30E-01 | 5.24E-01 | 6.75E-01 | 3.43E+00 | 6.75E-01 | 1.44E+01 |
| *Relative time* | 100% | 99% | 127% | 646% | 127% | 2725% |
| **Coq execution, `cbv`, Dense test** | | | | | | |
| *Time in s* | 1.89E-02 | 1.82E-02 | 1.92E-02 | 2.77E-02 | 3.05E-02 | 7.59E-01 |
| *Relative time* | 100% | 96% | 102% | 147% | 162% | 4025% |
| **Coq execution, `cbn`, Dense test** | | | | | | |
| *Time in s* | 5.05E+00 | 5.16E+00 | † | 5.77E+00 | 5.65E+00 | † |
| *Relative time* | 100% | 102% | † | 114% | 112% | † |
| **Extraction to OCaml, Sparse test** | | | | | | |
| *Time in s* | 1.18E-02 | 9.88E-03 | 7.68E-03 | 1.18E-02 | 1.68E-02 | 3.74E-02 |
| *Relative time* | 100% | 84% | 65% | 100% | 142% | 317% |
| *Allocated bytes* | 1.07E+07 | 8.11E+06 | 6.01E+06 | 1.07E+07 | 1.62E+07 | 1.04E+07 |
| *Relative allocated* | 100% | 76% | 56% | 100% | 151% | 97% |
| *Max memory usage* | 1.79E+07 | 1.16E+07 | 1.00E+07 | 1.79E+07 | 2.09E+07 | 1.02E+06 |
| *Relative memory usage* | 100% | 65% | 56% | 100% | 117% | 6% |
| **Extraction to OCaml, Dense test** | | | | | | |
| *Time in s* | 2.05E-04 | 1.94E-04 | 2.30E-04 | 2.06E-04 | 2.29E-04 | 3.08E-03 |
| *Relative time* | 100% | 95% | 112% | 100% | 112% | 1502% |
| *Allocated bytes* | 6.88E+05 | 5.98E+05 | 5.97E+05 | 6.88E+05 | 1.07E+06 | 3.91E+06 |
| *Relative allocated* | 100% | 87% | 87% | 100% | 156% | 568% |
| **Extraction to OCaml, Repeated test** | | | | | | |
| *Time in s* | 3.68E-02 | 3.62E-02 | 5.04E-02 | 3.72E-02 | 5.26E-02 | 6.27E-01 |
| *Relative time* | 100% | 98% | 137% | 101% | 143% | 1704% |
| *Allocated bytes* | 6.56E+08 | 5.44E+08 | 5.92E+08 | 6.56E+08 | 1.10E+09 | 1.78E+09 |
| *Relative allocated* | 100% | 83% | 90% | 100% | 168% | 271% |

**Table 1** Performance figures for 5 implementations of binary trees and for a reference AVL implementation. The implementations and the benchmark tests are described in the main text, section 7.1. The tests were executed on a single core of an AMD Ryzen 7 3700X processor running Ubuntu Linux 20.4, using Coq version 8.13.2 and OCaml version 4.12.0. All tests were repeated enough times so that the measured running time is between 1 and 10 seconds. Timing variations between multiple measurements are below 5%. For the Dense and Repeated tests, the maximal memory usage is too low to be meaningful. For `cbn` Coq executions, two entries were discarded (†): the Canonical implementation stops early with a result that is not in normal form, and the AVL implementation takes too long to evaluate.

native code using the `ocamlopt` compiler. For OCaml executions, we report execution times, amount of memory allocated, and maximal memory usage. For executions within Coq, we were unable to measure memory usage and only report execution times. The experimental results are listed in table 1.

We see that the Canonical implementation significantly improves on the Original implementation for the Sparse test: Coq execution times decrease by 20%, OCaml execution times by 33%, and memory allocations and memory usage are almost halved. For the Dense and Repeated tests, memory allocation is reduced by about 15%, but execution times increase a bit.

The Node01 implementation stands halfway between the Original and the Canonical implementations in terms of OCaml execution times and memory consumption. Coq execution times show no improvement compared with the Original implementation. Combined with

| *Implementation* | Original | Canonical | AVL | Trie |
|---|---|---|---|---|
| **Coq execution,** `vm_compute` | | | | |
| *Time in s* | 3.02E-02 | 2.67E-02 | 1.94E-01 | 3.82E-01 |
| *Relative time* | 100% | 89% | 641% | 1265% |
| **Extraction to OCaml** | | | | |
| *Time in s* | 1.50E-02 | 1.12E-02 | 6.55E-03 | 4.00E-03 |
| *Relative time* | 100% | 75% | 44% | 27% |
| *Allocated bytes* | 2.12E+07 | 1.65E+07 | 1.04E+07 | 8.73E+06 |
| *Relative allocated* | 100% | 78% | 49% | 41% |
| *Max memory usage* | 1.38E+07 | 7.48E+06 | 1.02E+06 | 4.74E+06 |
| *Relative memory usage* | 100% | 54% | 7% | 34% |

**Table 2** Performance figures for 4 implementations of the dictionary data structure. The implementations and the benchmark tests are described in the main text, section 7.2. The measurements use same the methodology and hardware and software configuration as in Table 1.

the lack of extensionality, these results show that Node01 is not an interesting alternative to Canonical.

The Sigma implementation performs exactly like the Original implementation after extraction to OCaml. This is unsurprising: the subset type is erased during extraction, resulting in very similar OCaml code for the Sigma and Original implementations. As expected, execution inside Coq is slowed down by the construction and propagation of proof terms for the well-formedness invariant: noticeably so (3% to 45% depending on the Coq evaluation mechanism used) for the Sparse and Dense tests, but dramatically so (a 6-times increase) for the Repeated test.

The GADT implementation avoids the dramatic degradations of the Sigma implementation, but runs 13% to 45% slower than the Original implementation, both within Coq and after extraction to OCaml. The extra arguments to the Node constructor increase memory usage by 17% and allocations by about 50%.

The AVL implementation performs poorly compared to the binary trie implementations: between 3 and 40 times slower than the Original implementation. AVL is a generic implementation of maps that, unlike the binary trie implementations, does not take advantage of the specific structure of keys as positive numbers. Instead, the AVL implementation spends much time in rebalancing and in comparing positive keys, a comparison that is quite slow since positive numbers are isomorphic to lists of bits.

## 7.2 Comparison with other dictionary data structures

CompCert and VST occasionally use binary tries as dictionaries, that is, finite maps indexed by character strings instead of by positive numbers. This is achieved by a simple encoding of strings as bit sequences represented as positive numbers. It is interesting to compare the performance of these dictionaries derived from binary tries with the performance of other implementations of dictionaries. Table 2 shows some measurements for the following 4 dictionary implementations:

**Original** and **Canonical** : These are two of the binary trie implementations used in section 7.1, composed with string to positive conversions. In other words, the get and set operations take strings as keys, convert them to positive numbers on the fly, and invoke the get and set operations of the Original or Canonical binary trie implementations.

**AVL**: The AVL balanced binary search trees from Coq's standard library, using strings as keys.

**CTrie**: A trie data structure that branches on characters, instead of on single bits like the binary tries. More precisely, each node of the trie carries a sparse, sorted association list mapping the next character to the corresponding sub-trie.

The benchmark used in table 2 is similar to the Sparse benchmark from section 7.1: 5064 words randomly chosen from an English dictionary are inserted in the data structure, then looked up.

For executions within Coq, the implementations based on binary tries are 6 to 14 times faster than the AVL and the CTrie implementations, despite the overhead of converting strings to positive numbers at each operation. This can be explained by the way Coq represents strings: a string is isomorphic to a list of characters, each character being isomorphic to a 8-tuple of Booleans. This makes comparisons between characters and comparisons between strings rather expensive. The AVL implementation performs many string comparisons, and the CTrie implementation performs may character comparisons.

After extraction to OCaml, Coq's characters are mapped to OCaml's characters, which are small machine integers that can be compared very efficiently. Strings remain as lists of small integers. Consequently, the AVL implementation runs (in OCaml) nearly twice as fast as the best binary trie-based implementation, and the CTrie implementation runs nearly three times faster. The performance of the dictionaries based on binary tries remains pretty good, given the simplicity of the implementation, and could be improved by "fusing" the string-to-positive conversion and the recursive tree traversals, avoiding the construction of positive numbers.

## 8 Substantial efficiency improvements in VST

Compared to the original binary tries, canonical binary tries *substantially* improve the efficiency of proofs in the Verified Software Toolchain (VST). VST [1] comprises a program logic (Verifiable C) for the C programming language, proved sound in Coq with respect to the operational semantics of CompCert Clight, and a proof automation system (VST-Floyd) [4] to assist users in applying the program logic to their program. Verifiable C and VST-Floyd use PTrees (CompCert's binary tries) very heavily,

- mapping function-names to function-bodies
- mapping function-names to their specifications
- mapping global variable names to their types
- mapping local variable names to their types
- mapping local variable names to their symbolic contents
- and so on.

The symbolic contents of a C local variable can be a Coq term that contains Coq variables bound in the local Coq context. Similarly, a function specification is a Coq term that contains shallowly embedded Coq expressions. That is, VST uses finite maps (tries) from C identifiers to shallowly embedded Coq terms that may have free Coq variables bound in the local Coq context. For those reasons, it is not possible to extract everything to OCaml and run it there; program proof in VST is conducted within Coq.

Therefore the binary tries that appear in VST theorem statements, VST proof terms, and VST-Floyd calculations must all be manipulated in (unextracted) Coq expressions. And

these are *concrete* tries—that is, a theorem statement might mention a finite map not just as an abstract variable, but a *particular* map binding identifiers `QuickSort`, `Partition`, `lobound`, `hibound`, `i`, `j`, *et cetera.* Thus, an actual *data structure* can appear in a theorem statement and be manipulated in Coq proofs. It is not only the *lookup* that is slow; very large tries occurring as subterms of theorem statements cause difficulties in proof-checking, even without many actual get operations. Differences in the representation efficiency of this structure get magnified when the entire data structure is (in effect) "quoted" in the Coq kernel, and traversed and transformed during Coq proof-checking. This is all the more reason to use asymptotically efficient data structures to represent finite maps.

Until recently, VST had only weak support for modular verification of modular programs, so most VST proofs were for a single .c file. In choosing positive keys to represent each of the identifiers in a .c file, we simply chose the first $n$ consecutive positive numbers. In this *short-idents* mode, the tries were fairly dense, and the paths traversed within the tree were fairly short. Therefore, until recently, CompCert's "original" binary tries had been efficient enough.

A recent addition to VST is *Verified Software Units* [2], a system for modular verification of modular C programs. The VSU system represents each software unit as a set of finite maps: function names to bodies, function names to specs, global variable names to types, global variable names to initializers, and so on. Symbolic linking of program units is performed using a combine operation to merge the tries.

Each .c file is the basis for one Verified Software Unit (VSU); it is augmented by specifications and proofs in Coq. In order to link one VSU to another, a given global name must be represented by the same positive key in each VSU. Therefore, in parsing a .c file into an Abstract Syntax Tree within Coq, we cannot simply choose the first $n$ consecutive positive numbers to represent the names *in that file*. And we would like to be able to link a VSU with .c programs yet to be written, in which we don't yet know what names will be used. So we now want to derive a name's positive key directly from the ASCII representation of the name. We construct an invertible function from ASCII strings to positive (using modest data compression, with about 6 bits per character on average).

But in this *canonical-idents* mode, we will have *sparse* binary tries: there will be many nodes in the tree whose "middle" component is None.

The VSU system needs to reason about the equivalence of, for example, the name-to-struct-type mapping used by two different modules that are being linked. Therefore, equality/equivalence of tries is a property frequently mentioned in theorem statements.

That is,

- VSU uses finite maps for more purposes, and maps with larger domains, than core VST;
- even with the same number of (nonempty) elements in a trie, path lengths are much longer in VSU than in core VST;
- VSU relies much more on equivalence properties in theorem statements than core VST.

Therefore, we might expect Canonical binary tries to improve the performance of VSU even more than they improve performance of core VST.

### 8.1 Measurements

*Core VST.* We ran VST on our standard test suite of 104 single-file C program verifications, in short-idents mode. VST runs within Coq; there are many insert, lookup, and fold operations on finite maps. The cost here is not only the computation of lookups by $\beta$-reduction, but

also the representation of particular concrete finite-map values within theorem statements and within proofs of other theorems. *Running our standard test suite of 104 C-program verifications, VST as a whole runs 8% faster with canonical binary tries versus simple binary tries (4018 seconds versus 4346).*

Given the measurements shown in section 7, this may seem surprising. In those measurements of a synthetic benchmark for update and lookup, Coq execution using `vm_compute` in the "Dense test" showed that canonical tries were, if anything, a bit slower than "original" tries. And even if the canonical tries themselves had been 8% or 20% faster, how could that lead to an 8% speedup of the entire VST system?

What's different about the VST execution is that *tries appear within theorem statements and proofs*, and *occasionally there are proofs about equivalence of tries*. We conjecture that the canonical tries' smaller representations, and the use of Leibnitz equality instead of equivalence, could make a major difference in proof size and proof-checking time.

*VSUs and linking.* We ran the *Verified Software Units* component of VST: a verification of several individual function-bodies and then the proof of VSU-linking into a whole verified program.[5] Here there are not only proofs of function-bodies, but also proofs of *combine* operations that simulate module linking. Again, the cost is not only $\beta$-reduction of *combine*, but the representation in theorem-statements and proofs of the resulting (and intermediate) finite-map terms, and equivalence tests of nontrivial tries. *The results are as follows:*

| VSU | dense (short idents) | sparse (canonical idents) |
|---|---|---|
| original binary tries | 270 sec | 296 |
| canonical binary tries | 101 | 199 |

Using canonical tries yields an enormous improvement—a factor of 1.6 to 2.7 gain in speed for the system overall. This is surprising! How could the representation of tries make such a difference?

Compared to core VST, the VSU system relies much more heavily on tries, the **combine** operator on tries, equivalence tests/proofs on tries; and all of these appearing in theorem-statements and proofs. The use of canonical tries may improve performance in three ways:

1. The cost of these manipulations, especially in the Coq kernel's proof checker, is significantly affected by the efficiency of the representations of finite maps, *especially* in proofs that are about the linking of VSUs, hence the merging of large finite maps.
2. Theorems proved with canonical binary tries can use Liebnitz equality, which is significantly simpler (in Coq proof terms) than the propagation of congruences on equivalence relations (necessary for nonextensional tries). VSU relies more on such equivalences than does core VST.
3. The *time* cost of these two affects can be superlinear in the space cost, if the Coq system is stressed in space and therefore frequently invoking the major-generation garbage collector.

So therefore: while it is inappropriate to generalize and say "one can expect a 1.6x to 2.7x speedup," we *can* say: Applications that build theorems and proofs about particular concrete finite maps are particularly likely to benefit from finite-map representations that are (1) efficient and (2) extensional.

[5] The program being verified is the "pile" example from Beringer and Appel [3].

## 9 Universe polymorphism

A logic in which Type is a member of Type is likely to contain paradoxes. Coq avoids paradoxes by stratifying Type into a countable infinity of universes, so that Type(i) is contained in Type(j) only if $i < j$. "Traditional" inductive types in Coq (such as positive or option) live at a particular universe level; the universe associated with (e.g.,) option is calculated not when option is declared, but after an entire proof development has accumulated a set of universe constraints $i < j$.

This can lead to the problem that a use of option $t_1$ in one part of a development, and option $t_2$ in another part, can lead to an unwanted universe constraint between two otherwise unrelated types $t_1$ and $t_2$. Recent versions of Coq have allowed inductive types to be declared *universe polymorphic,* which avoids this problem.

Our original PTree.tree type was not universe polymorphic, which led to problems in linking certain large proof developments. Declaring Polymorphic Inductive tree did not solve the problem, because it also uses the option type which is not universe-polymorphic. Although there is no obstacle in principle to making option polymorphic, in practice this is a widely used type and this change could affect many other parts of large proof developments.

Canonical tries can also be declared with Polymorphic Inductive tree', and these will be truly polymorphic because of the happenstance that they do not use the option type.

## 10 Conclusion

Our quest for finite maps that enjoy the extensionality property led us to a new data structure, *canonical binary tries,* that performs very well in many contexts: extracted to OCaml and compiled to native code, or represented in Coq proofs. In OCaml code, canonical binary trees reduce memory usage significantly and improve execution times somewhat compared with simple binary tries. When keys are character strings, other implementations such as balanced binary trees are faster, but they require primitive integers, whose associated implementations would then have to be trusted (or proved).

In functional programs written in Coq, canonical binary tries are also somewhat faster than simple binary tries. But when (concrete) tries must appear in Coq theorem statements and in Coq proofs, canonical tries are much more efficient, (probably) in part because the structures are smaller and in part because Leibnitz equality is more efficient than equivalence relations to reason about.

At first, we were worried about the increase in the number of cases for function definitions and for proofs, compared with the original binary tries where there are only two cases to consider. We demonstrated two approaches to synthesizing large case analyses instead of writing them by hand: one is based on Coq's tactic language and lets us fluidly combine synthesis-time case analysis, run-time case analysis, and synthesis-time symbolic analysis; the other is based on custom constructors and induction principles that lets us view the new data structure as the original, two-case data structure, combined with partial evaluation within Coq to eliminate the run-time overhead of the view. More generally, Coq offers powerful metaprogramming facilities that have great potential to facilitate not just the verification but also the implementation of complex data structures.

Besides binary tries, are there other useful data structures that admit canonical representations? And are these canonical representations always more memory-efficient than other representations? We have no general answers to these questions, but we can offer another

example of a canonical representation. Consider sets of positive numbers. The usual representation of sets as lists is neither canonical nor extensional: the set $\{1,2\}$ has several representations, $[1,2]$ or $[2,1]$ or $[1,1,2,1,2]$. We can recover extensionality by using a subset type $\{\ \mathsf{l} : \mathsf{list\ positive} \mid \mathsf{sorted\ l}\ \}$, where the sorted predicate ensures that the list is increasing and without repetitions. However, a canonical representation exists, as the list of (positive) differences from one set element to the next greater element. For instance, the set $\{1,4,9,11\}$ is uniquely represented by the list $[1,4-1,9-4,11-9]$, that is, $[1,3,5,2]$. This encoding is not only canonical, but also slighty more memory efficient than the usual sorted list representation, as the numbers stored in the list of differences are smaller than those stored in the sorted list. Can we play similar tricks for more complex data structures? We leave this question for future work.

## References


1. Andrew W. Appel. Verified Software Toolchain. In *20th European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.
2. Lennart Beringer. Verified software units. In *30th European Symposium on Programming (ESOP'21)*, volume 12648 of *LNCS*, pages 118–147. Springer, 2021.
3. Lennart Beringer and Andrew W. Appel. Abstraction and subsumption in modular verification of C programs. *Formal Methods in System Design*, 2021.
4. Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reason.*, 61(1-4):367–422, June 2018.
5. Robbert Krebbers et al. The coq-std++ extended standard library, module `stdpp.pmap`, 2012–2021. `https://plv.mpi-sws.org/coqdoc/stdpp/stdpp.pmap.html`.
6. Jean-Christophe Filliâtre and Pierre Letouzey. Functors for proofs and programs. In *13th European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.
7. Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL 2015: 42nd symposium Principles of Programming Languages*, pages 247–259. ACM, 2015.
8. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
9. Xavier Leroy and Damien Doligez. The CompCert verified C compiler, module `Maps`, 2005–2021. `https://compcert.org/doc-3.9/html/compcert.lib.Maps.html`.