Efficient, Extensional, and Generic Finite Maps in Coq-std++

Robbert Krebbers

Radboud University Nijmegen, The Netherlands
mail@robbertkrebbers.nl

Abstract

Finite maps are omnipresent in the formalization of programming languages in proof assistants. In this talk, I will present the `gmap` ("generic" map) implementation of finite maps in the Coq-std++ library. This implementation has recently been improved, and enjoys a number of interesting features. First, `gmap` is efficient—operations such as lookup/insert/delete have logarithmic time complexity, and union/intersection have linear time complexity. Second, `gmap` is extensional—maps are equal iff they are point-wise equal (without axioms). Third, `gmap` is generic in the type of keys. Fourth, `gmap` can be used in nested recursive definitions. The implementation of `gmap` is based on the "canonical" version of binary tries by Appel and Leroy, but generalized to become generic in the type of keys.

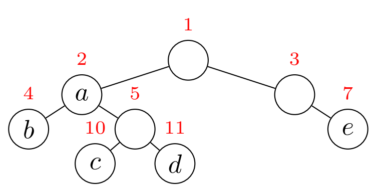
1 Introduction

A finite map with keys K and values A is a function $f : K \to \mathsf{option}\, A$ whose domain $\mathsf{dom}\, f$ is finite. Finite maps are widely used in the formalization of programming contexts—to represent heaps that map locations to values, typing contexts that map variables to types, or function bodies that map labels to statements. By taking A to be the unit type, one obtains finite sets, a similarly ubiquitous data structure.

Naively one could represent finite maps as association lists, *e.g.*, $[(2, a), (11, d)]$. In a proof assistant based on intentional type theory (without quotient types) such Coq, this approach allows for different representations of the same map. For example, the above map can also be represented as $[(11, d), (2, a)]$. Hence this naive representation does not satisfy the extensionality property, $m_1 = m_2$ iff $\forall k. m_1(k) = m_2(k)$. The lack of this property is a serious problem in large proof developments—one needs to reason up to setoid equality, and prove that all functions (including those defined by clients of the map library) respect setoid equality. The extensionality property is not satisfied by efficient map implementations such as AVL trees, Red-Black trees, B-trees either. It is therefore important to have a finite map representation that satisfies extensionality, is efficient, and generic in the keys.

2 Binary Tries

Appel and Leroy [1] show that binary tries are suitable for an efficient and extensional implementation of finite maps with `positive` keys. An example of a binary trie is (keys in red):



Since `positive` numbers in Coq are represented in binary, the basic map operations can be implemented by following the bit sequence. For example, 10 is represented as `xO (xI (xO xH))`. Starting at the root, one goes left/right/left to arrive at the value c of 10. Coq-std++ extends Leroy and Appel's work by making binary tries generic in the types of keys `K` using a type class to turn a key into a `positive` (*i.e.*, bit sequence):

```
Class Countable K `{EqDecision K} := {
  encode : K → positive;
  decode : positive → option K;
  decode_encode x : decode (encode x) = Some x
}.
```

(Coq-std++ provides `Countable` instances for the usual data types, such as numbers, sums, products, and lists. Inspired by ssreflect, Coq-std++ provides a `gen_tree` type to make it easy to define `Countable` instances for custom `Inductive` definitions.)

To implement the basic finite map operations on generic tries, we `encode` the key as a `positive`, and follow the bit sequence in the trie as described above.

To obtain the extensionality property ($m_1 = m_2$ iff $\forall k. m_1(k) = m_2(k)$) we need to ensure that tries are in canonical representation. Canonicity involves two key properties. First, there should be no subtrees with just empty nodes at the bottom. Second, every `positive` in the trie should be the result of `encode`. The old version of Coq-std++ used a Sigma-type to ensure that every trie is in canonical representation, but this approach has several problems (see Section 3). The new version uses the following representation:

```
Inductive gmap_dep_ne (A : Type) (P : positive → Prop) :=
  | GNode001 : gmap_dep_ne A (λ p, P p~1) → gmap_dep_ne A P
  | GNode010 : P 1 → A → gmap_dep_ne A P
  | GNode011 : P 1 → A → gmap_dep_ne A (λ p, P p~1) → gmap_dep_ne A P
  | GNode100 : gmap_dep_ne A (λ p, P p~0) → gmap_dep_ne A P
  | GNode101 : gmap_dep_ne A (λ p, P p~0) → gmap_dep_ne A (λ p, P p~1) → gmap_dep_ne A P
  | GNode110 : gmap_dep_ne A (λ p, P p~0) → P 1 → A → gmap_dep_ne A P
  | GNode111 : gmap_dep_ne A (λ p, P p~0) → P 1 → A → gmap_dep_ne A (λ p, P p~1) → gmap_dep_ne A P.

Inductive gmap_dep (A : Type) (P : positive → Prop) :=
  | GEmpty : gmap_dep A P
  | GNodes : gmap_dep_ne A P → gmap_dep A P.

Record gmap_key K `{Countable K} (q : positive) :=
  GMapKey { _ : encode (A:=K) <$> decode q = Some q }.

Record gmap K `{Countable K} A := GMap { gmap_car : gmap_dep A (gmap_key K) }.
```

Following Appel and Leroy [1] we define types for non-empty tries `gmap_dep_ne` and tries that might be empty `gmap_dep`. The constructors of `gmap_dep_ne` make sure that one cannot have subtrees with just empty nodes—namely, a node is only allowed to have no value if it has a non-empty child to the left or right.

Our new ingredient is the use of the predicate `P : positive → Prop`, which says that the key is "a valid encoding". At the top level (in the definition of `gmap`) we let `P` be `gmap_key K`, and in the constructors of `gmap_dep_ne` we make sure that `P` matches up with the position in the trie. Our representation is surprisingly easy to use. We can implement the operations for lookup, insert/delete/alter, mapping, merging, and folding without getting into any issues regarding dependent types.

3 Key Features

Up to our knowledge, Coq-std++'s `gmap` has a unique set of features that is not provided by any other Coq library for finite maps with generic keys. Most importantly:

- The extracted code is similar to handwritten code without dependent types because `P` is a `Prop`-based predicate and thus erased.
- Computation with `vm_compute` is efficient, and all equalities on closed maps hold definitionally. In the old Sigma-based version of Coq-std++ this was not the case because proofs were accumulated.
- Our maps can be used in nested recursive definitions, for example:

  ```
  Inductive gtest K `{Countable K} :=
    | GTest : gmap K (gtest K) → gtest K.
  ```

  With the old Sigma-type based definition Coq rejected this definition: the use of `gmap K (gtest K)` violates Coq's strict positivity condition. With our new definition these nested recursive definitions are accepted. One can nest things even further: `gtest K` is countable, allowing one to use `gtest` as keys in maps, *e.g.*, `gmap (gtest K) A` or `gtest (gtest K)`.

4 Coq Sources

The coq-std++ Gitlab can be found at `https://gitlab.mpi-sws.org/iris/stdpp`. The new `gmap` implementation can be at `https://gitlab.mpi-sws.org/iris/stdpp/-/blob/master/stdpp/gmap.v`. It is an instance of the `FinMap` type class, and to `gmap` one should always use the operations and theory of `FinMap`. The file `https://gitlab.mpi-sws.org/iris/stdpp/-/blob/master/stdpp/fin_maps.v` provides derived operations and lemmas for all implementations of `FinMap`.

References

[1] Andrew W. Appel and Xavier Leroy. Efficient extensional binary tries. *JAR*, 67(1):8, 2023.