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

Robbert Krebbers

Radboud University Nijmegen, The Netherlands

July 31, 2023 @ Coq Workshop, Białystok, Poland

Finite maps

Finite map / dictionary: Functions `K` $\rightarrow$ `option V` with finite support

Naive representation: Association lists

```
Definition map K V := list (K * V)
```

Example: `K:=string` and `V:=nat`

$$[(\text{`}\texttt{coq}\text{`}, 1989), (\text{`}\texttt{lean}\text{`}, 2013), (\text{`}\texttt{automath}\text{`}, 1967)]$$

Applications in programming languages

	Keys	**Values**
Heap (in high-level language)	Locations	Values
Heap (in machine language)	Addresses	Bytes
Function body	Labels	Statements
Typing context	Variables	Types
Sets over `A`	`A`	Unit

Wishlist

1. **Efficient operations:**
   - Logarithmic `lookup`/`insert`/`delete`, linear `union`/`intersection`
   - When extracted to OCaml <u>and</u> with `vm_compute` in Coq

Wishlist

1. **Efficient operations:**
   - Logarithmic `lookup`/`insert`/`delete`, linear `union`/`intersection`
   - When extracted to OCaml and with `vm_compute` in Coq
2. **Extensional equality (no setoids):**

$$m_1 = m_2 \quad \text{iff} \quad \forall k.\, m_1(k) = m_2(k)$$

Wishlist

1. **Efficient operations:**
   - Logarithmic `lookup`/`insert`/`delete`, linear `union`/`intersection`
   - When extracted to OCaml and with `vm_compute` in Coq
2. **Extensional equality (no setoids):**

$$m_1 = m_2 \quad \text{iff} \quad \forall k.\, m_1(k) = m_2(k)$$

Definitional extensional equality:
If m_1 and m_2 ground and $(\forall k.\, m_1(k) = m_2(k))$, then `eq_refl` : $m_1 = m_2$

Wishlist

1. **Efficient operations:**
   - Logarithmic `lookup`/`insert`/`delete`, linear `union`/`intersection`
   - When extracted to OCaml and with `vm_compute` in Coq
2. **Extensional equality (no setoids):**

   $$m_1 = m_2 \quad \text{iff} \quad \forall k.\, m_1(k) = m_2(k)$$

   Definitional extensional equality:
   If m_1 and m_2 ground and $(\forall k.\, m_1(k) = m_2(k))$, then `eq_refl` $: m_1 = m_2$
3. **Generic in the type of keys**

Wishlist

1. **Efficient operations:**
   - Logarithmic `lookup`/`insert`/`delete`, linear `union`/`intersection`
   - When extracted to OCaml and with `vm_compute` in Coq
2. **Extensional equality (no setoids):**

$$m_1 = m_2 \quad \text{iff} \quad \forall k.\, m_1(k) = m_2(k)$$

   Definitional extensional equality:
   If m_1 and m_2 ground and $(\forall k.\, m_1(k) = m_2(k))$, then `eq_refl` $: m_1 = m_2$
3. **Generic in the type of keys**
4. **Usable in nested inductive definitions:**

```
Inductive val :=
  | VInt : Z → val
  | VPair : val → val → val
  | VClosure : var → expr → map var val → val.
```

François Pottier 13:15

Hi! I seem to be running into a problem because Coq does not recognize that `gmap A B` is strictly positive in `B`. My use case is that I would like an environment to have type `gmap var val` (a map of variables to values) and I would like the values to include closures, which contain an environment. Has anyone run into this kind of problem before? Is there a workaround?

Coq does recognize that `Pmap_raw` is strictly positive, but that is two layers of abstraction below `gmap`...

Michael Sammler 🛠 13:22

This problem has been discussed before, but without finding a good solution, see e.g. the discussion here: https://mattermost.mpi-sws.org/iris/channels/stdpp/yz4br4wzspy47ckqpgahkwsnjh or here: https://mattermost.mpi-sws.org/iris/pl/jpjapa4ipf8nmp4m6irna4oqyw

François Pottier 13:27

Thanks! I guess I will use an association list instead (for now)...

Let us take a look at standard map representations

Comparison of map implementations

	assoc list
Efficient	○
Extensional	○
Generic	●
Nested induction	●

Problems with association lists

Inefficient: `lookup`/`insert`/`delete` are linear, `union`/`intersection` are quadratic

Problems with association lists

Inefficient: `lookup`/`insert`/`delete` are linear, `union`/`intersection` are quadratic

Extensionality fails: Order and duplicates matter

$$
\begin{aligned}
&[(\text{`}\mathtt{coq}\text{`}, 1989), (\text{`}\mathtt{lean}\text{`}, 2013), (\text{`}\mathtt{automath}\text{`}, 1967)] \\
\neq\ &[(\text{`}\mathtt{automath}\text{`}, 1967), (\text{`}\mathtt{coq}\text{`}, 1989), (\text{`}\mathtt{lean}\text{`}, 2013)]
\end{aligned}
$$

Problems with association lists

Inefficient: `lookup`/`insert`/`delete` are linear, `union`/`intersection` are quadratic

Extensionality fails: Order and duplicates matter

$$[(\text{`}\texttt{coq}\text{`}, 1989), (\text{`}\texttt{lean}\text{`}, 2013), (\text{`}\texttt{automath}\text{`}, 1967)]$$
$$\neq [(\text{`}\texttt{automath}\text{`}, 1967), (\text{`}\texttt{coq}\text{`}, 1989), (\text{`}\texttt{lean}\text{`}, 2013)]$$

Possible workarounds for extensionality:

- Use quotient type: Coq does not have those
- Use Σ type:

```
Definition map K V := { l : list (K * V) | sorted_by_key l }
```

  Breaks 'definitional extensional equality' (proofs are relevant) and 'usable in nested inductive definitions' (`map K V` not positive in `V`)

Comparison of map implementations (continued)

	assoc list	AVL
Efficient	○	●
Extensional	○	○
Generic	●	●
Nested induction	●	○

The standard efficient map representations (AVL, Red-Black, BTree) do not enjoy extensionality due to lack of quotient types in Coq

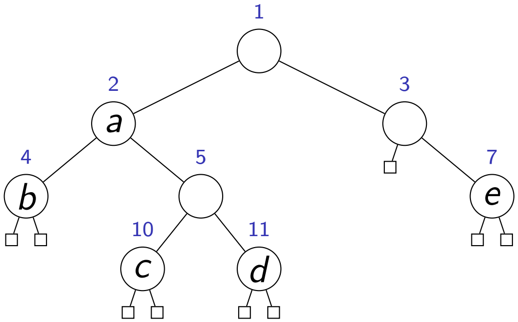
Need canonical representations

Binary tries [First version in Coq in CompCert, Leroy 2006]

```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A

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

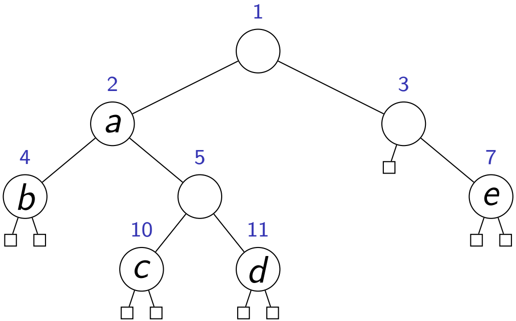
Binary tries [First version in Coq in CompCert, Leroy 2006]




```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A

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

Binary tries [First version in Coq in CompCert, Leroy 2006]

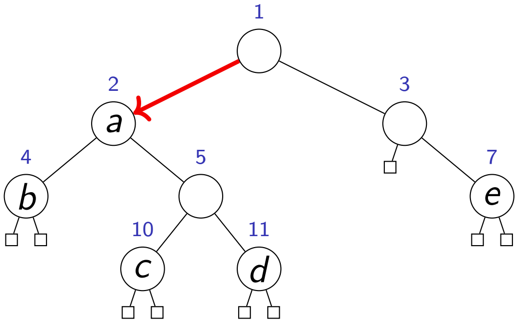



```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A

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

Example: Lookup for 10, in positive representation `xO (xI (xO xH))`

Binary tries [First version in Coq in CompCert, Leroy 2006]

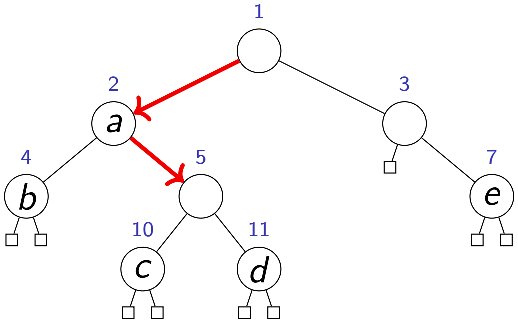



```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A

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

Example: Lookup for 10, in positive representation `xO (xI (xO xH))`

Binary tries [First version in Coq in CompCert, Leroy 2006]

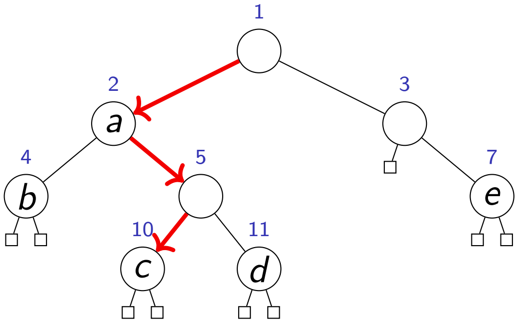



```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A

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

Example: Lookup for 10, in positive representation `xO (xI (xO xH))`

Binary tries [First version in Coq in CompCert, Leroy 2006]

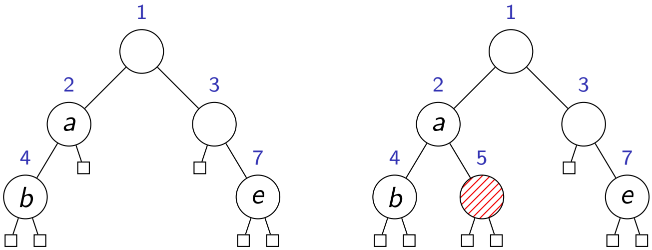



```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
               option A →
               trie A →
               trie A

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

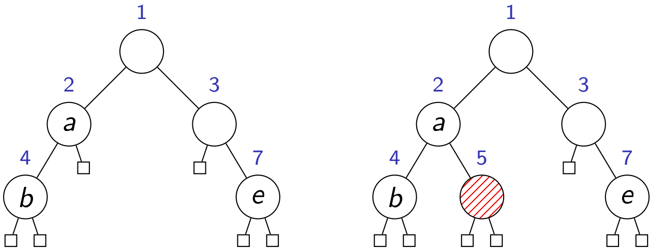
Example: Lookup for 10, in positive representation `xO (xI (xO xH))`

Extensionality for binary tries




```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A
```

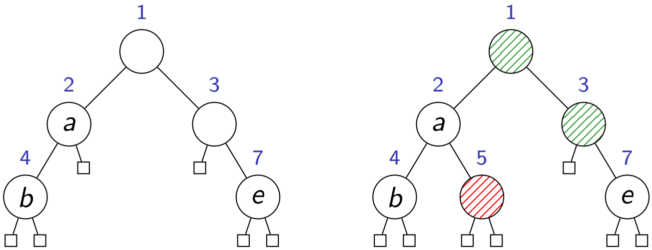
Extensionality for binary tries




```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A
```

Empty node invariant: A node can only be `None` if both subtrees are non-empty

Extensionality for binary tries




```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A
```

Empty node invariant: A node can only be `None` if both subtrees are non-empty

Generic keys [std++ 2012, inspired by ssreflect's countType]

Generalize from `positive` **to any** `K` **with** `Countable K`**:**

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

Generic keys [std++ 2012, inspired by ssreflect's countType]

Generalize from `positive` to any `K` with `Countable K`:

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

Examples:

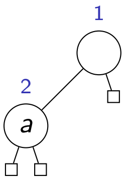
true	1
false	2

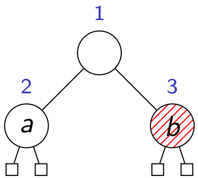
inl a	x0 (encode a)
inr b	xI (encode b)

(a,b)	a0 (b0 .. (an (bn xH))) where a0 .. (an xH) = encode a and b0 .. (bn xH) = encode b

Extensionality for generic tries

Let `K := bool` and `encode b := if b then 1 else 2`

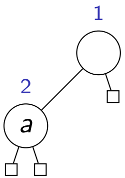


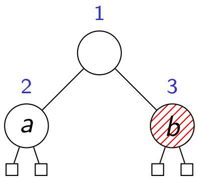



```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A
```

Extensionality for generic tries

Let `K := bool` and `encode b := if b then 1 else 2`






```
Inductive trie A :=
  | Leaf : trie A
  | Node : trie A →
           option A →
           trie A →
           trie A
```

Valid key invariant: A node is only `Some` if its key `q` is a valid code, *i.e.*,

`encode <$> decode q = Some q`

Extensional generic tries using Σ type [std++ 2012–2022]

```
Inductive Pmap (A : Type) := PMap {
  pmap_car : trie A;
  pmap_prf : Pmap_wf pmap_car (* Non-empty node invariant *)
}.

Record gmap (K : Type) `{Countable K} (A : Type) := GMap {
  gmap_car : Pmap A;
  gmap_prf : gmap_wf K gmap_car (* Valid key invariant *)
}.
```

Comparison of map implementations (continued)

	assoc list	AVL	old `gmap`
Efficient	○	●	●
Extensional	○	○	◐
Generic	●	●	●
Nested induction	●	○	○

Be aware of Σ types!

Problems with Σ types in Coq

```
Inductive Pmap (A : Type) := PMap {
  pmap_car : trie A;
  pmap_prf : Pmap_wf pmap_car (* Non-empty node invariant *)
}.
```

Problems with Σ types in Coq

```
Inductive Pmap (A : Type) := PMap {
  pmap_car : trie A;
  pmap_prf : Pmap_wf pmap_car (* Non-empty node invariant *)
}.
```

1. To avoid computation of proofs, inhabitants of `Pmap_wf` need to opaque
   This destroys definitional extensional equality

```
Lemma foo : delete 10 {[10:=12]} =@{Pmap Z} ∅
Proof. Fail reflexivity. (* Unable to unify "delete 10 .." with "∅". *) Qed.
```

Problems with Σ types in Coq

```
Inductive Pmap (A : Type) := PMap {
  pmap_car : trie A;
  pmap_prf : Pmap_wf pmap_car (* Non-empty node invariant *)
}.
```

1. To avoid computation of proofs, inhabitants of `Pmap_wf` need to opaque
   This destroys <u>definitional extensional equality</u>

   ```
   Lemma foo : delete 10 {[10:=12]} =@{Pmap Z} ∅
   Proof. Fail reflexivity. (* Unable to unify "delete 10 .." with "∅". *) Qed.
   ```

2. **They destroy positivity checking in <u>nested inductive definitions</u>**

   ```
   Fail Inductive val :=
     | VInt : Z → val
     | VPair : val → val → val
     | VClosure : var → expr → Pmap val → val.
   (* Non strictly positive occurrence of "val" in ".. → Pmap val → ..". *)
   ```

Inspiration for this work

Journal of Automated Reasoning (2023) 67:8
https://doi.org/10.1007/s10817-022-09655-x

Efficient Extensional Binary Tries

Andrew W. Appel[1] · Xavier Leroy[2]

Received: 10 October 2021 / Accepted: 3 October 2022 / Published online: 12 January 2023

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

Comparison of map implementations (continued)

	assoc list	AVL	old `gmap`	App/Ler
Efficient	○	●	●	●
Extensional	○	○	◐	●
Generic	●	●	●	○
Nested induction	●	○	○	●

Extensional tries without Σ type [Appel/Leroy, 2023]

Key idea: Enumerate all valid shapes of nodes as constructors
⇒ ensures non-empty node invariant by construction

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

Inductive trie (A : Type) :=
  | Empty : trie A
  | Nodes : ne_trie A → trie A.
```

Comparison of map implementations (continued)

This work ↓

	assoc list	AVL	old gmap	App/Ler	new gmap
Efficient	○	●	●	●	●
Extensional	○	○	◐	●	●
Generic	●	●	●	○	●
Nested induction	●	○	○	●	●

Challenge for supporting generic keys

Key challenge: Define valid key invariant without Σ type around the whole tree
Solution: Dependent/indexed types

- Ensure that all the operations and proofs can be done without pain
  $\Rightarrow$ Use the 'right' definition, smart constructor, case analysis, induction principle
- Extraction to OCaml should give the Appel/Leroy definition
  $\Rightarrow$ Put index of dependent type in `Prop`

The data structure

```
Inductive gmap_dep_ne (A : Type) (P : positive → Prop) :=
  ...
```

The index `P : positive → Prop` expresses if the key is valid

- At the top level `P q := encode <$> decode q = Some q`
- Propagate in tree using:

```
Notation "P ~ 0" := (λ p, P (xO p)) : function_scope.
Notation "P ~ 1" := (λ p, P (xI p)) : function_scope.
```

- Since `P` has sort `Prop` it is erased by extraction

Full definition of the data structure

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

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

Full definition of the data structure

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

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

(* Wrapped in a Record to avoid evaluation of encode/decode *)
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) }.
```

Implementation of lookup

```
Definition gmap_dep_ne_lookup {A} : ∀ {P}, positive → gmap_dep_ne A P → option A :=
  fix go {P} i t {struct t} :=
  match t, i with
  | (GNode010 _ x|GNode011 _ x _|GNode110 _ _ x|GNode111 _ _ x _), 1 => Some x
  | (GNode100 l|GNode110 l _ _|GNode101 l _|GNode111 l _ _ _), i~0 => go i l
  | (GNode001 r|GNode011 _ _ r|GNode101 _ r|GNode111 _ _ _ r), i~1 => go i r
  | _, _ => None
  end.
```

Implementation of lookup

```
Definition gmap_dep_ne_lookup {A} : ∀ {P}, positive → gmap_dep_ne A P → option A :=
  fix go {P} i t {struct t} :=
  match t, i with
  | (GNode010 _ x|GNode011 _ x _|GNode110 _ _ x|GNode111 _ _ x _), 1 => Some x
  | (GNode100 l|GNode110 l _ _|GNode101 l _|GNode111 l _ _ _), i~0 => go i l
  | (GNode001 r|GNode011 _ _ r|GNode101 _ r|GNode111 _ _ _ r), i~1 => go i r
  | _, _ => None
  end.
```

Take away: Dependent pattern matching ‘just’ works

Handling many cases

Problem: To implement operations such as `union` you get $7^2 = 49$ cases

Handling many cases

Problem: To implement operations such as `union` you get $7^2 = 49$ cases

Inspired by Appel/Leroy we provide:

Handling many cases

Problem: To implement operations such as `union` you get $7^2 = 49$ cases

Inspired by Appel/Leroy we provide:

- **Smart constructor**

```
GNode : gmap_dep A P~0 →
        option (P 1 * A) →
        gmap_dep A P~1 → gmap_dep A P
```

Handling many cases

Problem: To implement operations such as `union` you get $7^2 = 49$ cases

Inspired by Appel/Leroy we provide:

- **Smart constructor**

```
GNode : gmap_dep A P~0 →
        option (P 1 * A) →
        gmap_dep A P~1 → gmap_dep A P
```

- **Case analysis**

```
gmap_dep_ne_case : gmap_dep_ne A P →
                   (gmap_dep A P~0 → option (P 1 * A) → gmap_dep A P~1 → B) →
                   B
```

Handling many cases

Problem: To implement operations such as `union` you get $7^2 = 49$ cases

Inspired by Appel/Leroy we provide:

- **Smart constructor**

```
GNode : gmap_dep A P~0 →
        option (P 1 * A) →
        gmap_dep A P~1 → gmap_dep A P
```

- **Case analysis**

```
gmap_dep_ne_case : gmap_dep_ne A P →
                   (gmap_dep A P~0 → option (P 1 * A) → gmap_dep A P~1 → B) →
                   B
```

- **Induction principle**

Result: The entire std++ `FinMap` interface can be implemented and verified in 503 LOC

(including imports and some comments)

Result: The entire std++ `FinMap` interface can be implemented and verified in 503 LOC

(including imports and some comments)

No need for `eq_rect` or axioms

Comparison of map implementations (continued)

This work ↓ (new `gmap`)

	assoc list	AVL	old `gmap`	App/Ler	new `gmap`	math-comp
Efficient	○	●	●	●	●	○
Extensional	○	○	◐	●	●	◐
Generic	●	●	●	○	●	●
Nested induction	●	○	○	●	●	●

Comparison with finmaps in math-comp

```
Structure finSet (K : choiceType) : Type := mkFinSet {
  enum_fset :> seq K;
  _ : canonical_keys enum_fset
}.
Record finMap (K : choiceType) (V : Type) : Type := FinMap {
  domf : {fset K};
  ffun_of_fmap :> {ffun domf -> V}
}.
```

Comparison with finmaps in math-comp

```
Structure finSet (K : choiceType) : Type := mkFinSet {
  enum_fset :> seq K;
  _ : canonical_keys enum_fset
}.
Record finMap (K : choiceType) (V : Type) : Type := FinMap {
  domf : {fset K};
  ffun_of_fmap :> {ffun domf -> V}
}.
```

- ○ Sets as lists, coding using `nat`, so not very efficient
- ● Finite functions `{ffun ..}` have been defined so that nested induction works, see `https://github.com/math-comp/math-comp/pull/294`
- ◐ No definitional extensional equality due to Σ type in `finSet`

Future work

- Definitional extensionality would work with Σ type in `SProp`
  - Challenge: `SProp` is not very well integrated in Coq's stdlib
  - Use in nested inductives still a problem

Future work

- Definitional extensionality would work with Σ type in `SProp`
  - Challenge: `SProp` is not very well integrated in Coq's stdlib
  - Use in nested inductives still a problem
- What if future versions of Coq have quotients (or HITs)
  - Opens door for more map implementations: AVL, RedBlack, *etc.*
  - Extensionality will be no problem
  - Use in nested inductives is unclear, what about positivity restrictions on quotients/HITs

Future work

- Definitional extensionality would work with Σ type in `SProp`
  - Challenge: `SProp` is not very well integrated in Coq's stdlib
  - Use in nested inductives still a problem
- What if future versions of Coq have quotients (or HITs)
  - Opens door for more map implementations: AVL, RedBlack, *etc.*
  - Extensionality will be no problem
  - Use in nested inductives is unclear, what about positivity restrictions on quotients/HITs
- Proper benchmarking
  - Appel and Leroy have benchmarks for `lookup`/`insert`
  - For those, Appel/Leroy are factor 1.5-5 faster
    (conjecture of problem: our `insert` is not native, but defined in terms of
    `partial_alter : (option A → option A) → K → gmap K A → gmap K A`)
  - Need good benchmarks for other map operations (*e.g.*, `union`)

Advertisement: Other features of std++

- Type classes for operator and property overloading
- Type classes for properties of types (decidable, finite, countable, infinite, ...)
- Theory and derived operations on maps
- Theory and operations on lists
- Sets, finite sets, finite multisets
- Tactics: `naive_solver`, `set_solver`, `multiset_solver`

`https://gitlab.mpi-sws.org/iris/stdpp`