

Tries

Andreas Lochbihler and Tobias Nipkow

May 6, 2022

Abstract

This article formalizes the “trie” data structure invented by Fredkin [1]. It also provides a specialization where the entries in the trie are lists.

Contents

1	Trie	1
1.1	Empty trie	3
1.2	<i>lookup-trie</i>	3
1.3	<i>delete-trie</i>	4
1.4	<i>update-with-trie</i>	4
1.5	Domain of a trie	5
1.6	Range of a trie	5
2	Tries (List Version)	5
2.1	<i>lookup-tries</i>	6
2.2	<i>insert-tries, inserts-tries, tries-of-list</i>	6
2.3	<i>set-tries</i>	7

1 Trie

```
theory Trie
imports HOL-Library.AList
begin
```

```
datatype ('k, 'v) trie =
  Trie 'v option ('k * ('k, 'v)trie) list
```

```
lemma trie-induct [case-names Trie, induct type]:
  ( $\bigwedge vo kvs. (\bigwedge k t. (k, t) \in set kvs \implies P t) \implies P (Trie vo kvs)$ )  $\implies P t$ 
  <proof>
```

definition *empty-trie* :: ('k, 'v) trie **where**
empty-trie = Trie None []

fun *is-empty-trie* :: ('k, 'v) trie \Rightarrow bool **where**
is-empty-trie (Trie v m) = (v = None \wedge m = [])

fun *lookup-trie* :: ('k, 'v) trie \Rightarrow 'k list \Rightarrow 'v option **where**
lookup-trie (Trie v m) [] = v |
lookup-trie (Trie v m) (k#ks) =
 (case map-of m k of
 None \Rightarrow None |
 Some st \Rightarrow *lookup-trie* st ks)

fun *update-with-trie* ::
 'k list \Rightarrow ('v option \Rightarrow 'v) \Rightarrow ('k, 'v) trie \Rightarrow ('k, 'v) trie **where**
update-with-trie [] f (Trie v ps) = Trie (Some(f v)) ps |
update-with-trie (k#ks) f (Trie v ps) =
 Trie v (AList.update-with-aux *empty-trie* k (*update-with-trie* ks f) ps)

The function argument *f* of *update-with-trie* does not return an optional value because *None* could break the invariant that no empty tries are contained in a trie because *AList.update-with-aux* cannot recognise and remove empty tries. Therefore the delete function is implemented separately rather than via *update-with-trie*.

Do not use *update-with-trie* if most of the calls do not change the entry (because of the garbage this creates); use *lookup-trie* possibly followed by *update-trie*. This shortcoming could be addressed if *f* indicated that the entry is unchanged, eg by *None*.

definition *update-trie* :: 'k list \Rightarrow 'v \Rightarrow ('k, 'v) trie \Rightarrow ('k, 'v) trie **where**
update-trie ks v = *update-with-trie* ks (%-. v)

lemma *update-trie-induct*:
 $\llbracket \bigwedge v ps. P [] (Trie v ps); \bigwedge k ks v ps. (\bigwedge x. P ks x) \implies P (k\#ks) (Trie v ps) \rrbracket \implies$
 $P xs t$
 <proof>

lemma *update-trie-Nil[simp]*: *update-trie* [] v (Trie vo ts) = Trie (Some v) ts
 <proof>

lemma *update-trie-Cons[simp]*: *update-trie* (k#ks) v (Trie vo ts) =
 Trie vo (AList.update-with-aux (Trie None []) k (*update-trie* ks v) ts)
 <proof>

fun *delete-trie* :: 'key list \Rightarrow ('key, 'val) trie \Rightarrow ('key, 'val) trie
where
delete-trie [] (Trie vo ts) = Trie None ts |

```

delete-trie (k#ks) (Trie vo ts) =
  (case map-of ts k of
    None  $\Rightarrow$  Trie vo ts |
    Some t  $\Rightarrow$  let t' = delete-trie ks t
                 in if is-empty-trie t'
                     then Trie vo (AList.delete-aux k ts)
                     else Trie vo (AList.update k t' ts))

```

```

fun all-trie :: ('v  $\Rightarrow$  bool)  $\Rightarrow$  ('k, 'v) trie  $\Rightarrow$  bool where
all-trie p (Trie v ps) =
  ((case v of None  $\Rightarrow$  True | Some v  $\Rightarrow$  p v)  $\wedge$  ( $\forall$  (k,t)  $\in$  set ps. all-trie p t))

```

```

fun invar-trie :: ('key, 'val) trie  $\Rightarrow$  bool where
invar-trie (Trie vo kts) =
  (distinct (map fst kts)  $\wedge$ 
   ( $\forall$  (k, t)  $\in$  set kts.  $\neg$  is-empty-trie t  $\wedge$  invar-trie t))

```

1.1 Empty trie

```

lemma invar-empty [simp]: invar-trie empty-trie
<proof>

```

```

lemma is-empty-conv: is-empty-trie ts  $\longleftrightarrow$  ts = Trie None []
<proof>

```

1.2 lookup-trie

```

lemma lookup-empty [simp]: lookup-trie empty-trie = Map.empty
<proof>

```

```

lemma lookup-empty' [simp]: lookup-trie (Trie None []) ks = None
<proof>

```

```

lemma lookup-update:
  lookup-trie (update-trie ks v t) ks' = (if ks = ks' then Some v else lookup-trie t
  ks')
<proof>

```

```

lemma lookup-update':
  lookup-trie (update-trie ks v t) = (lookup-trie t)(ks  $\mapsto$  v)
<proof>

```

```

lemma lookup-eq-Some-iff:
assumes invar: invar-trie ((Trie vo kus) :: ('key, 'val) trie)
shows lookup-trie (Trie vo kus) ks = Some v  $\longleftrightarrow$ 
  (ks = []  $\wedge$  vo = Some v)  $\vee$ 
  ( $\exists$  k t ks'. ks = k # ks'  $\wedge$ 
   (k, t)  $\in$  set kus  $\wedge$  lookup-trie t ks' = Some v)
<proof>

```

lemma *lookup-eq-None-iff*:
assumes *invar*: *invar-trie* ((*Trie vo kvs*) :: ('key, 'val) *trie*)
shows *lookup-trie* (*Trie vo kvs*) *ks* = *None* \longleftrightarrow
 (*ks* = [] \wedge *vo* = *None*) \vee
 ($\exists k ks'$. *ks* = *k* # *ks'* \wedge ($\forall t$. (*k*, *t*) \in *set kvs* \longrightarrow *lookup-trie t ks'* = *None*))
 <proof>

lemma *update-not-empty*: \neg *is-empty-trie* (*update-trie ks v t*)
 <proof>

lemma *invar-trie-update*: *invar-trie t* \implies *invar-trie* (*update-trie ks v t*)
 <proof>

lemma *is-empty-lookup-empty*:
invar-trie t \implies *is-empty-trie t* \longleftrightarrow *lookup-trie t* = *Map.empty*
 <proof>

lemma *lookup-update-with-trie*:
lookup-trie (*update-with-trie ks f t*) *ks'* =
 (if *ks'* = *ks* then *Some*(*f*(*lookup-trie t ks'*)) else *lookup-trie t ks'*)
 <proof>

1.3 delete-trie

lemma *delete-eq-empty-lookup-other-fail*:
 [*delete-trie ks t* = *Trie None* []; *ks'* \neq *ks*] \implies *lookup-trie t ks'* = *None*
 <proof>

lemma *lookup-delete*: *invar-trie t* \implies
lookup-trie (*delete-trie ks t*) *ks'* =
 (if *ks* = *ks'* then *None* else *lookup-trie t ks'*)
 <proof>

lemma *lookup-delete'*:
invar-trie t \implies *lookup-trie* (*delete-trie ks t*) = (*lookup-trie t*)(*ks* := *None*)
 <proof>

lemma *invar-trie-delete*:
invar-trie t \implies *invar-trie* (*delete-trie ks t*)
 <proof>

1.4 update-with-trie

lemma *nonempty-update-with-aux*: *AList.update-with-aux v k f ps* \neq []
 <proof>

lemma *nonempty-update-with-trie*: \neg *is-empty-trie* (*update-with-trie ks f t*)
 <proof>

lemma *invar-update-with-trie*:

invar-trie $t \implies \text{invar-trie } (\text{update-with-trie } ks \ f \ t)$
 <proof>

1.5 Domain of a trie

lemma *dom-lookup*:

$\text{dom } (\text{lookup-trie } (\text{Trie } vo \ kts)) =$
 $(\bigcup k \in \text{dom } (\text{map-of } kts). \text{Cons } k \ ' \ \text{dom } (\text{lookup-trie } (\text{the } (\text{map-of } kts \ k)))) \cup$
 $(\text{if } vo = \text{None} \ \text{then } \{\} \ \text{else } \{\}\})$
 <proof>

lemma *finite-dom-lookup*:

$\text{finite } (\text{dom } (\text{lookup-trie } t))$
 <proof>

lemma *dom-lookup-empty-conv*: $\text{invar-trie } t \implies \text{dom } (\text{lookup-trie } t) = \{\} \iff$
is-empty-trie t
 <proof>

1.6 Range of a trie

lemma *ran-lookup-Trie*: $\text{invar-trie } (\text{Trie } vo \ ps) \implies$

$\text{ran } (\text{lookup-trie } (\text{Trie } vo \ ps)) =$
 $(\text{case } vo \ \text{of } \text{None} \Rightarrow \{\} \mid \text{Some } v \Rightarrow \{v\}) \cup (\bigcup (k,t) : \text{set } ps. \text{ran}(\text{lookup-trie } t))$
 <proof>

lemma *all-trie-eq-ran*:

$\text{invar-trie } t \implies \text{all-trie } P \ t = (\forall x \in \text{ran}(\text{lookup-trie } t). \ P \ x)$
 <proof>

end

2 Tries (List Version)

theory *Tries*

imports *Trie*

begin

This is a specialization of tries where values are lists.

type-synonym $('k, 'v)\text{tries} = ('k, 'v \ \text{list})\text{trie}$

definition *lookup-tries* :: $('k, 'v)\text{tries} \Rightarrow 'k \ \text{list} \Rightarrow 'v \ \text{list}$ **where**
 $\text{lookup-tries } t \ ks = (\text{case } \text{lookup-trie } t \ ks \ \text{of } \text{None} \Rightarrow [] \mid \text{Some } vs \Rightarrow vs)$

definition *update-with-tries* ::

$'k \ \text{list} \Rightarrow ('v \ \text{list} \Rightarrow 'v \ \text{list}) \Rightarrow ('k, 'v) \ \text{tries} \Rightarrow ('k, 'v) \ \text{tries}$ **where**
 $\text{update-with-tries } ks \ f =$
 $\text{update-with-trie } ks \ (\lambda vo. \ \text{case } vo \ \text{of } \text{None} \Rightarrow f \ [] \mid \text{Some } vs \Rightarrow f \ vs)$

definition *insert-tries* :: 'k list \Rightarrow 'v \Rightarrow ('k,'v)tries \Rightarrow ('k,'v)tries **where**
insert-tries ks v =
 update-with-trie ks (λ vo. case vo of None \Rightarrow [v] | Some vs \Rightarrow v#vs)

definition *inserts-tries* :: ('v \Rightarrow 'k list) \Rightarrow 'v list \Rightarrow ('k,'v)tries \Rightarrow ('k,'v)tries
where
inserts-tries key = fold (%v. *insert-tries* (key v) v)

definition *tries-of-list* :: ('v \Rightarrow 'k list) \Rightarrow 'v list \Rightarrow ('k,'v)tries **where**
tries-of-list key vs = *inserts-tries* key vs (Trie None [])

definition *set-tries* :: ('k,'v)tries \Rightarrow 'v set **where**
set-tries t = Union {gs. \exists a. gs = set(lookup-tries t a)}

definition *all-tries* :: ('v \Rightarrow bool) \Rightarrow ('k,'v)tries \Rightarrow bool **where**
all-tries P = *all-trie* (list-all P)

2.1 lookup-tries

lemma *lookup-Nil[simp]*:
 lookup-tries (Trie vo ps) [] = (case vo of None \Rightarrow [] | Some vs \Rightarrow vs)
 <proof>

lemma *lookup-Cons[simp]*: lookup-tries (Trie vo ps) (a#as) =
 (case map-of ps a of None \Rightarrow [] | Some at \Rightarrow lookup-tries at as)
 <proof>

lemma *lookup-empty[simp]*: lookup-tries (Trie None []) as = []
 <proof>

theorem *lookup-update*:
 lookup-tries (update-trie as vs t) bs =
 (if as=bs then vs else lookup-tries t bs)
 <proof>

theorem *lookup-update-with*:
 lookup-tries (update-with-tries as f t) bs =
 (if as=bs then f(lookup-tries t as) else lookup-tries t bs)
 <proof>

2.2 insert-tries, inserts-tries, tries-of-list

lemma *invar-insert-tries*: invar-trie t \Longrightarrow invar-trie(insert-tries as v t)
 <proof>

lemma *invar-inserts-tries*:
 invar-trie t \Longrightarrow invar-trie (inserts-tries key xs t)
 <proof>

lemma *invar-of-list*: invar-trie (tries-of-list key xs)

$\langle proof \rangle$

lemma *set-lookup-insert-tries*: $set (lookup-tries (insert-tries ks a t) ks') =$
 $(if ks' = ks then Set.insert a (set(lookup-tries t ks')) else set(lookup-tries t ks'))$
 $\langle proof \rangle$

lemma *in-set-lookup-inserts-tries*:
 $(v \in set(lookup-tries (inserts-tries key vs t) (key v))) =$
 $(v \in set vs \cup set(lookup-tries t (key v)))$
 $\langle proof \rangle$

lemma *in-set-lookup-of-list*:
 $v \in set(lookup-tries (tries-of-list key vs) (key v)) = (v \in set vs)$
 $\langle proof \rangle$

lemma *in-set-lookup-inserts-triesD*:
 $v \in set(lookup-tries (inserts-tries key vs t) xs) \implies$
 $v \in set vs \cup set(lookup-tries t xs)$
 $\langle proof \rangle$

lemma *in-set-lookup-of-listD*:
 $v \in set(lookup-tries (tries-of-list f vs) xs) \implies v \in set vs$
 $\langle proof \rangle$

2.3 set-tries

lemma *set-tries-eq-ran*: $set-tries t = Union(set ' ran(lookup-trie t))$
 $\langle proof \rangle$

lemma *set-tries-empty[simp]*: $set-tries (Trie None []) = \{\}$
 $\langle proof \rangle$

lemma *set-tries-insert[simp]*:
 $set-tries (insert-tries a x t) = Set.insert x (set-tries t)$
 $\langle proof \rangle$

lemma *set-insert-tries*:
 $set-tries (inserts-tries key xs t) =$
 $set xs \cup set-tries t$
 $\langle proof \rangle$

lemma *set-tries-of-list[simp]*:
 $set-tries(tries-of-list key xs) = set xs$
 $\langle proof \rangle$

lemma *in-set-lookup-set-triesD*:
 $x \in set (lookup-tries t a) \implies x \in set-tries t$
 $\langle proof \rangle$

end

References

- [1] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.