

Tries

Andreas Lochbihler and Tobias Nipkow

February 23, 2021

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>	5
1.4	<i>update-with-trie</i>	8
1.5	Domain of a trie	8
1.6	Range of a trie	10
2	Tries (List Version)	10
2.1	<i>lookup-tries</i>	11
2.2	<i>insert-tries, inserts-tries, tries-of-list</i>	11
2.3	<i>set-tries</i>	12

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]:
  ( $\bigwedge vo\ kvs. (\bigwedge k\ t. (k, t) \in set\ kvs \implies P\ t) \implies P\ (Trie\ vo\ kvs)$ )  $\implies P\ t$ 
by induction-schema (pat-completeness, lexicographic-order)
```

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*:
 $\llbracket \bigwedge v ps. P \rrbracket$ (Trie v ps); $\bigwedge k ks v ps. (\bigwedge x. P ks x) \implies P (k\#ks) (Trie v ps) \implies$
 $P xs t$
by *induction-schema* (*pat-completeness*, *lexicographic-order*)

lemma *update-trie-Nil[simp]*: *update-trie* [] v (Trie vo ts) = Trie (Some v) ts
by(*simp add: update-trie-def*)

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)
by(*simp add: update-trie-def empty-trie-def*)

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
by(simp add: empty-trie-def)

```

```

lemma is-empty-conv: is-empty-trie ts  $\longleftrightarrow$  ts = Trie None []
by(cases ts)(simp)

```

1.2 lookup-trie

```

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

```

```

  fix ks show lookup-trie empty-trie ks = Map.empty ks
  by(cases ks)(auto simp add: empty-trie-def)
qed

```

```

lemma lookup-empty' [simp]: lookup-trie (Trie None []) ks = None
by(simp add: lookup-empty[unfolded empty-trie-def])

```

lemma lookup-update:

```

lookup-trie (update-trie ks v t) ks' = (if ks = ks' then Some v else lookup-trie t ks')

```

```

proof(induct ks t arbitrary: ks' rule: update-trie-induct)

```

```

  case (1 vo ts ks')
  show ?case by(fastforce simp add: neq-Nil-conv dest: not-sym)

```

next

```

  case (2 k ks vo ts ks')
  show ?case by(cases ks')(auto simp add: map-of-update-with-aux 2 split: option.split)

```

qed

lemma lookup-update':

```

lookup-trie (update-trie ks v t) = (lookup-trie t)(ks  $\mapsto$  v)

```

by(*rule ext*)(*simp add: lookup-update*)

lemma *lookup-eq-Some-iff*:

assumes *invar: invar-trie* ((*Trie vo kvs*) :: ('key, 'val) *trie*)

shows *lookup-trie* (*Trie vo kvs*) *ks* = *Some v* \longleftrightarrow

(*ks* = [] \wedge *vo* = *Some v*) \vee

($\exists k t ks'. ks = k \# ks' \wedge$

(*k, t*) \in *set kvs* \wedge *lookup-trie t ks'* = *Some v*)

proof (*cases ks*)

case Nil thus ?*thesis* **by** *simp*

next

case (*Cons k ks'*)

note *ks-eq[simp]* = *Cons*

show ?*thesis*

proof (*cases map-of kvs k*)

case None thus ?*thesis*

apply (*simp*)

apply (*auto simp add: map-of-eq-None-iff image-iff Ball-def*)

done

next

case (*Some t'*) **note** *map-eq* = *this*

from *invar* **have** *dist-kvs: distinct* (*map fst kvs*) **by** *simp*

from *map-of-eq-Some-iff*[*OF dist-kvs, of k*] *map-eq*

show ?*thesis* **by** *simp metis*

qed

qed

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

($\exists k ks'. ks = k \# ks' \wedge (\forall t. (k, t) \in \text{set } kvs \longrightarrow \text{lookup-trie } t \text{ } ks' = \text{None})$)

using *lookup-eq-Some-iff*[*of vo kvs ks, OF invar*]

apply (*cases ks*)

apply *auto*[]

apply (*auto split: option.split*)

apply (*metis option.simps(3) weak-map-of-SomeI*)

apply (*metis option.exhaust*)

apply (*metis option.exhaust*)

done

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

apply(*cases t*)

apply(*rename-tac kvs*)

apply(*cases ks*)

apply(*case-tac [2] kvs*)

apply (*auto*)

done

lemma *invar-trie-update*: $\text{invar-trie } t \implies \text{invar-trie } (\text{update-trie } ks \ v \ t)$
by(*induct* $ks \ t$ *rule*: *update-trie-induct*)(*auto simp add*: *set-update-with-aux update-not-empty split*: *option.splits*)

lemma *is-empty-lookup-empty*:
 $\text{invar-trie } t \implies \text{is-empty-trie } t \iff \text{lookup-trie } t = \text{Map.empty}$

proof(*induct* t)
case (*Trie* $vo \ kvs$)
thus *?case*
apply(*cases* kvs)
apply(*auto simp add*: *fun-eq-iff elim*: *allE*[**where** $x = []$])
apply(*erule meta-allE*)
apply(*erule meta-impE*)
apply(*rule disjI1*)
apply(*fastforce intro*: *exI*[**where** $x = a \# b$ **for** $a \ b$])
done
qed

lemma *lookup-update-with-trie*:
 $\text{lookup-trie } (\text{update-with-trie } ks \ f \ t) \ ks' =$
(if $ks' = ks$ *then* $\text{Some}(f(\text{lookup-trie } t \ ks'))$ *else* $\text{lookup-trie } t \ ks'$)

proof(*induction* $ks \ t$ *arbitrary*: ks' *rule*: *update-trie-induct*)
case 1 **thus** *?case* **by**(*auto simp add*: *neq-Nil-conv*)
next
have $*$: $\bigwedge xs \ y \ ys. (xs \neq y \# ys) = (xs = [] \vee (\exists x \ zs. xs = x \# zs \wedge (x \neq y \vee zs \neq ys)))$
by *auto* (*metis neq-Nil-conv*)
case 2
thus *?case* **by**(*auto simp*: $*$ *map-of-update-with-aux split*: *option.split*)
qed

1.3 delete-trie

lemma *delete-eq-empty-lookup-other-fail*:
 $\llbracket \text{delete-trie } ks \ t = \text{Trie None } []; ks' \neq ks \rrbracket \implies \text{lookup-trie } t \ ks' = \text{None}$

proof(*induct* $ks \ t$ *arbitrary*: ks' *rule*: *delete-trie.induct*)
case 1 **thus** *?case* **by**(*auto simp add*: *neq-Nil-conv*)
next
case ($2 \ k \ ks \ vo \ ts$)
show *?case*
proof(*cases* *map-of* $ts \ k$)
case (*Some* t)
show *?thesis*
proof(*cases* ks')
case (*Cons* $k' \ ks''$)
show *?thesis*
proof(*cases* $k' = k$)
case *False*

```

from Some Cons 2.prem1 have AList.delete-aux k ts = []
  by(clarsimp simp add: Let-def split: if-split-asm)
with False have map-of ts k' = None
by(cases map-of ts k')(auto dest: map-of-SomeD simp add: delete-aux-eq-Nil-conv)
thus ?thesis using False Some Cons 2.prem1 by simp
next
  case True
  with Some 2.prem1 Cons show ?thesis
    by(clarsimp simp add: 2.hyps Let-def is-empty-conv split: if-split-asm)
  qed
qed(insert Some 2.prem1, simp add: Let-def split: if-split-asm)
next
  case None thus ?thesis using 2.prem1 by simp
  qed
qed

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(induct ks t arbitrary: ks' rule: delete-trie.induct)
  case 1 show ?case by(fastforce dest: not-sym simp add: neq-Nil-conv)
next
  case (2 k ks vo ts)
  show ?case
  proof(cases ks')
    case Nil thus ?thesis by(simp split: option.split add: Let-def)
  next
    case [simp]: (Cons k' ks'')
    show ?thesis
    proof(cases k' = k)
      case False thus ?thesis using 2.prem1
      by(auto simp add: Let-def update-conv' map-of-delete-aux split: option.split)
    next
      case [simp]: True
      show ?thesis
      proof(cases map-of ts k)
        case None thus ?thesis by simp
      next
        case (Some t)
        thus ?thesis
        proof(cases is-empty-trie (delete-trie ks t))
          case True
          with Some 2.prem1 show ?thesis
          by(auto simp add: map-of-delete-aux is-empty-conv dest: delete-eq-empty-lookup-other-fail)
        next
          case False
          thus ?thesis using Some 2 by(auto simp add: update-conv')
        qed
      qed
    qed
  qed

```

qed
 qed
 qed

lemma *lookup-delete'*:

invar-trie t \implies *lookup-trie (delete-trie ks t) = (lookup-trie t)(ks := None)*
by(rule ext)(simp add: lookup-delete)

lemma *invar-trie-delete*:

invar-trie t \implies *invar-trie (delete-trie ks t)*
proof(induct ks t rule: delete-trie.induct)

case 1 thus ?case by simp

next

case (2 k ks vo ts)

show ?case

proof(cases map-of ts k)

case None

thus ?thesis using 2.prem by simp

next

case (Some t)

with 2.prem have *invar-trie t* by auto

with Some have *invar-trie (delete-trie ks t)* by (rule 2)

from 2.prem Some have *distinct: distinct (map fst ts) \neg is-empty-trie t* by auto

show ?thesis

proof(cases *is-empty-trie (delete-trie ks t)*)

case True

{ fix k' t'

assume *k't'*: $(k', t') \in \text{set } (AList.delete\text{-aux } k \text{ } ts)$

with *distinct* have *map-of (AList.delete-aux k ts) k' = Some t'* by simp

hence *map-of ts k' = Some t'* using *distinct*

by (auto

simp del: *map-of-eq-Some-iff*

simp add: *map-of-delete-aux*

split: *if-split-asm*)

with 2.prem have \neg *is-empty-trie t' \wedge invar-trie t'* by auto }

with 2.prem have *invar-trie (Trie vo (AList.delete-aux k ts))* by auto

thus ?thesis using True Some by (simp)

next

case False

{ fix k' t'

assume *k't'*: $(k', t') \in \text{set } (AList.update\ k \ (delete\text{-trie } ks \ t) \ ts)$

hence *map-of (AList.update k (delete-trie ks t) ts) k' = Some t'*

using 2.prem by (auto simp add: *distinct-update*)

hence *eq: ((map-of ts)(k \mapsto delete-trie ks t)) k' = Some t'* unfolding

update-conv .

have \neg *is-empty-trie t' \wedge invar-trie t'*

proof(cases *k' = k*)

case True

```

    with eq have t' = delete-trie ks t by simp
    with ⟨invar-trie (delete-trie ks t)⟩ False
    show ?thesis by simp
  next
    case False
    with eq distinct have (k', t') ∈ set ts by simp
    with 2.prem1 show ?thesis by auto
  qed }
  thus ?thesis using Some 2.prem1 False by(auto simp add: distinct-update)
qed
qed
qed

```

1.4 update-with-trie

lemma *nonempty-update-with-aux*: $AList.update-with-aux\ v\ k\ f\ ps \neq []$
by (*induction ps*) *auto*

lemma *nonempty-update-with-trie*: $\neg is-empty-trie (update-with-trie\ ks\ f\ t)$
by(*induction ks t rule: update-trie-induct*)
(auto simp: nonempty-update-with-aux)

lemma *invar-update-with-trie*:
invar-trie t \implies invar-trie (update-with-trie ks f t)
by(*induction ks f t rule: update-with-trie.induct*)
(auto simp: set-update-with-aux nonempty-update-with-trie split: option.split prod.splits)

1.5 Domain of a trie

lemma *dom-lookup*:
 $dom (lookup-trie (Trie\ vo\ kts)) =$
 $(\bigcup k \in dom (map-of\ kts). Cons\ k\ 'dom (lookup-trie (the (map-of\ kts\ k)))) \cup$
 $(if\ vo = None\ then\ \{\}\ else\ \{\})$
unfolding *dom-def*
apply(*rule sym*)
apply(*safe*)
apply *simp*
apply(*clarisimp simp add: if-split-asm*)
apply(*case-tac x*)
apply(*auto split: option.split-asm*)
done

lemma *finite-dom-lookup*:
 $finite (dom (lookup-trie\ t))$
proof(*induct t*)
case (*Trie vo kts*)
have $finite (\bigcup k \in dom (map-of\ kts). Cons\ k\ 'dom (lookup-trie (the (map-of\ kts\ k))))$
proof(*rule finite-UN-I*)


```

  show finite (dom (map-of kts)) by(rule finite-dom-map-of)
next
fix k
assume k ∈ dom (map-of kts)
then obtain v where (k, v) ∈ set kts map-of kts k = Some v by(auto dest:
map-of-SomeD)
hence finite (dom (lookup-trie (the (map-of kts k)))) by simp(rule Trie)
thus finite (Cons k ' dom (lookup-trie (the (map-of kts k)))) by(rule fi-
nite-imageI)
qed
thus ?case by(simp add: dom-lookup)
qed

```

```

lemma dom-lookup-empty-conv: invar-trie t ⇒ dom (lookup-trie t) = {} ↔
is-empty-trie t
proof(induct t)
case (Trie vo kvs)
show ?case
proof
assume dom: dom (lookup-trie (Trie vo kvs)) = {}
have vo = None
proof(cases vo)
case (Some v)
hence [] ∈ dom (lookup-trie (Trie vo kvs)) by auto
with dom have False by simp
thus ?thesis ..
qed
moreover have kvs = []
proof(cases kvs)
case (Cons kt kvs')
with ⟨invar-trie (Trie vo kvs)⟩
have ¬ is-empty-trie (snd kt) invar-trie (snd kt) by auto
from Cons have (fst kt, snd kt) ∈ set kvs by simp
hence dom (lookup-trie (snd kt)) = {} ↔ is-empty-trie (snd kt)
using ⟨invar-trie (snd kt)⟩ by(rule Trie)
with ⟨¬ is-empty-trie (snd kt)⟩ have dom (lookup-trie (snd kt)) ≠ {} by simp
with dom Cons have False by(auto simp add: dom-lookup)
thus ?thesis ..
qed
ultimately show is-empty-trie (Trie vo kvs) by simp
next
assume is-empty-trie (Trie vo kvs)
thus dom (lookup-trie (Trie vo kvs)) = {}
by(simp add: lookup-empty[unfolded empty-trie-def])
qed
qed

```

1.6 Range of a trie

lemma *ran-lookup-Trie*: $\text{invar-trie } (Trie \text{ vo } ps) \implies$
 $\text{ran } (\text{lookup-trie } (Trie \text{ vo } ps)) =$
 $(\text{case vo of None} \Rightarrow \{\} \mid \text{Some } v \Rightarrow \{v\}) \cup (UN (k,t) : \text{set } ps. \text{ran}(\text{lookup-trie } t))$
by(*auto simp add: ran-def lookup-eq-Some-iff split: prod.splits option.split*)

lemma *all-trie-eq-ran*:
 $\text{invar-trie } t \implies \text{all-trie } P \ t = (\forall x \in \text{ran}(\text{lookup-trie } t). P \ x)$
by(*induction P t rule: all-trie.induct*)
(auto simp add: ran-lookup-Trie split: prod.splits option.split)

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**
 $\text{lookup-tries } t \ ks = (\text{case lookup-trie } t \ ks \text{ of None} \Rightarrow [] \mid \text{Some } vs \Rightarrow vs)$

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

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

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

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

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

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

2.1 lookup-tries

lemma *lookup-Nil*[simp]:

$lookup-tries (Trie\ vo\ ps) [] = (case\ vo\ of\ None \Rightarrow [] \mid Some\ vs \Rightarrow vs)$
by (*simp add: lookup-tries-def*)

lemma *lookup-Cons*[simp]: $lookup-tries (Trie\ vo\ ps) (a\#\ as) =$

$(case\ map-of\ ps\ a\ of\ None \Rightarrow [] \mid Some\ at \Rightarrow lookup-tries\ at\ as)$
by (*simp add: lookup-tries-def split: option.split*)

lemma *lookup-empty*[simp]: $lookup-tries (Trie\ None\ [])\ as = []$

by(*case-tac as, simp-all add: lookup-tries-def*)

theorem *lookup-update*:

$lookup-tries (update-trie\ as\ vs\ t)\ bs =$
 $(if\ as=bs\ then\ vs\ else\ lookup-tries\ t\ bs)$

by(*auto simp add: lookup-tries-def lookup-update*)

theorem *lookup-update-with*:

$lookup-tries (update-with-tries\ as\ f\ t)\ bs =$
 $(if\ as=bs\ then\ f(lookup-tries\ t\ as)\ else\ lookup-tries\ t\ bs)$

by(*auto simp add: lookup-tries-def update-with-tries-def lookup-update-with-trie split: option.split*)

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

lemma *invar-insert-tries*: $invar-trie\ t \Longrightarrow invar-trie(insert-tries\ as\ v\ t)$

by(*simp add: insert-tries-def invar-update-with-trie split: option.split*)

lemma *invar-inserts-tries*:

$invar-trie\ t \Longrightarrow invar-trie (inserts-tries\ key\ xs\ t)$

by(*induct xs arbitrary: t*)(*auto simp: invar-insert-tries inserts-tries-def*)

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

by(*simp add: tries-of-list-def invar-inserts-tries*)

lemma *set-lookup-insert-tries*: $set (lookup-tries (insert-tries\ ks\ a\ t)\ ks') =$

$(if\ ks' = ks\ then\ Set.insert\ a\ (set(lookup-tries\ t\ ks'))\ else\ set(lookup-tries\ t\ ks'))$

by(*simp add: lookup-tries-def insert-tries-def lookup-update-with-trie set-eq-iff split: option.split*)

lemma *in-set-lookup-inserts-tries*:

$(v \in set(lookup-tries (inserts-tries\ key\ vs\ t) (key\ v))) =$
 $(v \in set\ vs \cup set(lookup-tries\ t (key\ v)))$

by(*induct vs arbitrary: t*)

(*auto simp add: inserts-tries-def set-lookup-insert-tries*)

lemma *in-set-lookup-of-list*:

$v \in set(lookup-tries (tries-of-list\ key\ vs) (key\ v)) = (v \in set\ vs)$

by(*simp add: tries-of-list-def in-set-lookup-inserts-tries*)

lemma *in-set-lookup-inserts-triesD*:
 $v \in \text{set}(\text{lookup-tries } (\text{inserts-tries key vs } t) \text{ } xs) \implies$
 $v \in \text{set } vs \cup \text{set}(\text{lookup-tries } t \text{ } xs)$
apply (*induct vs arbitrary: t*)
apply (*simp add: inserts-tries-def*)
apply (*simp add: inserts-tries-def*)
apply (*fastforce simp add: set-lookup-insert-tries split: if-splits*)
done

lemma *in-set-lookup-of-listD*:
 $v \in \text{set}(\text{lookup-tries } (\text{tries-of-list } f \text{ } vs) \text{ } xs) \implies v \in \text{set } vs$
by (*auto simp: tries-of-list-def dest: in-set-lookup-inserts-triesD*)

2.3 *set-tries*

lemma *set-tries-eq-ran*: $\text{set-tries } t = \text{Union}(\text{set } \text{' } \text{ran}(\text{lookup-trie } t))$
apply (*auto simp add: set-eq-iff set-tries-def lookup-tries-def ran-def*)
apply *metis*
by (*metis option.inject*)

lemma *set-tries-empty[*simp*]*: $\text{set-tries } (\text{Trie None } []) = \{\}$
by (*simp add: set-tries-def*)

lemma *set-tries-insert[*simp*]*:
 $\text{set-tries } (\text{insert-tries } a \text{ } x \text{ } t) = \text{Set.insert } x \text{ } (\text{set-tries } t)$
apply (*auto simp: set-tries-def lookup-update set-lookup-insert-tries*)
by (*metis insert-iff*)

lemma *set-insert-tries*:
 $\text{set-tries } (\text{inserts-tries key } xs \text{ } t) =$
 $\text{set } xs \text{ } \text{Un } \text{set-tries } t$
by (*induct xs arbitrary: t*) (*auto simp: inserts-tries-def*)

lemma *set-tries-of-list[*simp*]*:
 $\text{set-tries}(\text{tries-of-list key } xs) = \text{set } xs$
by (*simp add: tries-of-list-def set-insert-tries*)

lemma *in-set-lookup-set-triesD*:
 $x \in \text{set } (\text{lookup-tries } t \text{ } a) \implies x \in \text{set-tries } t$
by (*auto simp: set-tries-def*)

end

References

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