

Formalizing the Logic-Automaton Connection

Markus Reiter Stefan Berghofer

December 7, 2022

Abstract

This work presents a formalization of a library for automata on bit strings. It forms the basis of a reflection-based decision procedure for Presburger arithmetic, which is efficiently executable thanks to Isabelle’s code generator. With this work, we therefore provide a mechanized proof of a well-known connection between logic and automata theory. The formalization is also described in a publication [1].

Contents

1	General automata	1
2	BDDs	3
3	DFAs	7
3.1	Minimization	8
4	NFAs	39
5	Automata Constructions	45
5.1	Negation	45
5.2	Product Automaton	45
5.3	Transforming DFAs to NFAs	57
5.4	Transforming NFAs to DFAs	61
5.5	Quantifiers	73
5.6	Right Quotient	79
5.7	Diophantine Equations	84
5.8	Diophantine Inequations	99
6	Presburger Arithmetic	103
1	General automata	
	definition	

$reach\ tr\ p\ as\ q = (q = foldl\ tr\ p\ as)$

lemma *reach-nil*: $reach\ tr\ p\ []\ p$ **by** (*simp add: reach-def*)

lemma *reach-snoc*: $reach\ tr\ p\ bs\ q \implies reach\ tr\ p\ (bs\ @\ [b])\ (tr\ q\ b)$
by (*simp add: reach-def*)

lemma *reach-nil-iff*: $reach\ tr\ p\ []\ q = (p = q)$ **by** (*auto simp add: reach-def*)

lemma *reach-snoc-iff*: $reach\ tr\ p\ (bs\ @\ [b])\ k = (\exists\ q. reach\ tr\ p\ bs\ q \wedge k = tr\ q\ b)$
by (*auto simp add: reach-def*)

lemma *reach-induct* [*consumes 1, case-names Nil snoc, induct set: reach*]:
assumes $reach\ tr\ p\ w\ q$
and $P\ []\ p$
and $\bigwedge k\ x\ y. \llbracket reach\ tr\ p\ x\ k; P\ x\ k \rrbracket \implies P\ (x\ @\ [y])\ (tr\ k\ y)$
shows $P\ w\ q$
using *assms* **by** (*induct w arbitrary: q rule: rev-induct*) (*simp add: reach-def*)+

lemma *reach-trans*: $\llbracket reach\ tr\ p\ a\ r; reach\ tr\ r\ b\ q \rrbracket \implies reach\ tr\ p\ (a\ @\ b)\ q$
by (*simp add: reach-def*)

lemma *reach-inj*: $\llbracket reach\ tr\ p\ a\ q; reach\ tr\ p\ a\ q' \rrbracket \implies q = q'$
by (*simp add: reach-def*)

definition
 $accepts\ tr\ P\ s\ as = P\ (foldl\ tr\ s\ as)$

locale *Automaton* =
fixes $trans :: 'a \Rightarrow 'b \Rightarrow 'a$
and $is-node :: 'a \Rightarrow bool$
and $is-alpha :: 'b \Rightarrow bool$
assumes $trans-is-node: \bigwedge q\ a. \llbracket is-node\ q; is-alpha\ a \rrbracket \implies is-node\ (trans\ q\ a)$
begin

lemma *steps-is-node*:
assumes $is-node\ q$
and $list-all\ is-alpha\ w$
shows $is-node\ (foldl\ trans\ q\ w)$
using *assms* **by** (*induct w arbitrary: q*) (*simp add: trans-is-node*)+

lemma *reach-is-node*: $\llbracket reach\ trans\ p\ w\ q; is-node\ p; list-all\ is-alpha\ w \rrbracket \implies is-node\ q$
by (*simp add: steps-is-node reach-def*)

end

2 BDDs

definition

is-alph :: *nat* \Rightarrow *bool list* \Rightarrow *bool* **where**
is-alph *n* = (λw . *length* *w* = *n*)

datatype *'a bdd* = *Leaf 'a* | *Branch 'a bdd 'a bdd* **for** *map: bdd-map*

primrec *bddh* :: *nat* \Rightarrow *'a bdd* \Rightarrow *bool*

where

bddh *n* (*Leaf* *x*) = *True*
| *bddh* *n* (*Branch* *l r*) = (*case* *n* of 0 \Rightarrow *False* | *Suc* *m* \Rightarrow *bddh* *m* *l* \wedge *bddh* *m* *r*)

lemma *bddh-ge*:

assumes *m* \geq *n*

assumes *bddh* *n* *bdd*

shows *bddh* *m* *bdd*

using *assms*

proof (*induct bdd arbitrary: n m*)

case (*Branch* *l r*)

then obtain *v* **where** *V: n = Suc v* **by** (*cases* *n*) *simp+*

show *?case* **proof** (*cases* *n = m*)

case *True*

with *Branch* **show** *?thesis* **by** *simp*

next

case *False*

with *Branch* **have** $\exists w$. *m = Suc w* \wedge *n* \leq *w* **by** (*cases* *m*) *simp+*

then obtain *w* **where** *W: m = Suc w* \wedge *n* \leq *w* **..**

with *Branch* *V* **have** *v* \leq *w* \wedge *bddh* *v* *l* \wedge *bddh* *v* *r* **by** *simp*

with *Branch* **have** *bddh* *w* *l* \wedge *bddh* *w* *r* **by** *blast*

with *W* **show** *?thesis* **by** *simp*

qed

qed *simp*

abbreviation *bdd-all* \equiv *pred-bdd*

fun *bdd-lookup* :: *'a bdd* \Rightarrow *bool list* \Rightarrow *'a*

where

bdd-lookup (*Leaf* *x*) *bs* = *x*

| *bdd-lookup* (*Branch* *l r*) (*b#bs*) = *bdd-lookup* (*if* *b* *then* *r* *else* *l*) *bs*

lemma *bdd-all-bdd-lookup*: \llbracket *bddh* (*length* *ws*) *bdd*; *bdd-all* *P* *bdd* $\rrbracket \Longrightarrow$ *P* (*bdd-lookup* *bdd* *ws*)

by (*induct bdd ws rule: bdd-lookup.induct*) *simp+*

lemma *bdd-all-bdd-lookup-iff*: *bddh* *n* *bdd* \Longrightarrow *bdd-all* *P* *bdd* = (\forall *ws*. *length* *ws* = *n* \longrightarrow *P* (*bdd-lookup* *bdd* *ws*))

apply (*rule iffI*)

apply (*simp add: bdd-all-bdd-lookup*)

```

proof (induct bdd arbitrary: n)
  case Leaf thus ?case
    apply simp
    apply (erule mp)
    apply (rule-tac x=replicate n False in exI, simp)
    done
next
  case (Branch l r n)
  then obtain k where k: n = Suc k by (cases n) simp+
  from Branch have R:  $\bigwedge ws. \text{length } ws = n \implies P (\text{bdd-lookup } (\text{Branch } l \ r) \ ws)$ 
by simp
  have  $\bigwedge ws. \text{length } ws = k \implies P (\text{bdd-lookup } l \ ws) \wedge P (\text{bdd-lookup } r \ ws)$ 
  proof -
    fix ws :: bool list assume H: length ws = k
    with k have length (False#ws) = n by simp
    hence 1:  $P (\text{bdd-lookup } (\text{Branch } l \ r) (\text{False}\#ws))$  by (rule R)
    from H k have length (True#ws) = n by simp
    hence  $P (\text{bdd-lookup } (\text{Branch } l \ r) (\text{True}\#ws))$  by (rule R)
    with 1 show  $P (\text{bdd-lookup } l \ ws) \wedge P (\text{bdd-lookup } r \ ws)$  by simp
  qed
  with Branch k show ?case by auto
qed

```

```

lemma bdd-all-bdd-map:
  assumes bdd-all P bdd
  and  $\bigwedge a. P a \implies Q (f a)$ 
  shows bdd-all Q (bdd-map f bdd)
  using assms by (induct bdd) simp+

```

```

lemma bddh-bdd-map:
  shows bddh n (bdd-map f bdd) = bddh n bdd
proof
  assume bddh n (bdd-map f bdd) thus bddh n bdd proof (induct bdd arbitrary:
n)
    case (Branch l r n)
    then obtain k where n = Suc k by (cases n) simp+
    with Branch show ?case by simp
  qed simp
next
  assume bddh n bdd thus bddh n (bdd-map f bdd) proof (induct bdd arbitrary:
n)
    case (Branch l r n)
    then obtain k where n = Suc k by (cases n) simp+
    with Branch show ?case by simp
  qed simp
qed

```

```

lemma bdd-map-bdd-lookup:
  assumes bddh (length ws) bdd

```

shows $\text{bdd-lookup } (\text{bdd-map } f \text{ bdd}) \text{ ws} = f (\text{bdd-lookup } \text{ bdd } \text{ ws})$
using *assms* **by** (*induct* *bdd ws rule: bdd-lookup.induct*) (*auto simp add: bddh-bdd-map*)⁺

fun *bdd-binop* :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a bdd \Rightarrow 'b bdd \Rightarrow 'c bdd

where

$\text{bdd-binop } f (\text{Leaf } x) (\text{Leaf } y) = \text{Leaf } (f \ x \ y)$
 $| \text{bdd-binop } f (\text{Branch } l \ r) (\text{Leaf } y) = \text{Branch } (\text{bdd-binop } f \ l (\text{Leaf } y)) (\text{bdd-binop } f \ r (\text{Leaf } y))$
 $| \text{bdd-binop } f (\text{Leaf } x) (\text{Branch } l \ r) = \text{Branch } (\text{bdd-binop } f (\text{Leaf } x) \ l) (\text{bdd-binop } f (\text{Leaf } x) \ r)$
 $| \text{bdd-binop } f (\text{Branch } l_1 \ r_1) (\text{Branch } l_2 \ r_2) = \text{Branch } (\text{bdd-binop } f \ l_1 \ l_2) (\text{bdd-binop } f \ r_1 \ r_2)$

lemma *bddh-binop*: $\text{bddh } n (\text{bdd-binop } f \ l \ r) = (\text{bddh } n \ l \ \wedge \ \text{bddh } n \ r)$

by (*induct* *f l r arbitrary: n rule: bdd-binop.induct*) (*auto split: nat.split-asm*)

lemma *bdd-lookup-binop*: $\llbracket \text{bddh } (\text{length } \text{bs}) \ l; \ \text{bddh } (\text{length } \text{bs}) \ r \rrbracket \Longrightarrow \text{bdd-lookup } (\text{bdd-binop } f \ l \ r) \ \text{bs} = f (\text{bdd-lookup } l \ \text{bs}) (\text{bdd-lookup } r \ \text{bs})$

apply (*induct* *f l r arbitrary: bs rule: bdd-binop.induct*)

apply *simp*

apply (*case-tac* *bs*)

apply *simp+*

apply (*case-tac* *bs*)

apply *simp+*

apply (*case-tac* *bs*)

apply *simp+*

done

lemma *bdd-all-bdd-binop*:

assumes *bdd-all P bdd*

and *bdd-all Q bdd'*

and $\bigwedge a \ b. \llbracket P \ a; \ Q \ b \rrbracket \Longrightarrow R \ (f \ a \ b)$

shows *bdd-all R (bdd-binop f bdd bdd')*

using *assms* **by** (*induct* *f bdd bdd' rule: bdd-binop.induct*) *simp+*

lemma *insert-list-idemp[simp]*:

$\text{List.insert } x (\text{List.insert } x \ \text{xs}) = \text{List.insert } x \ \text{xs}$

by *simp*

primrec *add-leaves* :: 'a bdd \Rightarrow 'a list \Rightarrow 'a list

where

$\text{add-leaves } (\text{Leaf } x) \ \text{xs} = \text{List.insert } x \ \text{xs}$

$| \text{add-leaves } (\text{Branch } b \ c) \ \text{xs} = \text{add-leaves } c \ (\text{add-leaves } b \ \text{xs})$

lemma *add-leaves-bdd-lookup*:

$\text{bddh } n \ b \Longrightarrow (x \in \text{set } (\text{add-leaves } b \ \text{xs})) = ((\exists \text{bs}. x = \text{bdd-lookup } b \ \text{bs} \ \wedge \ \text{is-alph } n \ \text{bs}) \vee x \in \text{set } \text{xs})$

apply (*induct* *b arbitrary: xs n*)

apply (*auto split: nat.split-asm*)

```

apply (rule-tac x=replicate n arbitrary in exI)
apply (simp add: is-alph-def)
apply (rule-tac x=True # bs in exI)
apply (simp add: is-alph-def)
apply (rule-tac x=False # bs in exI)
apply (simp add: is-alph-def)
apply (case-tac bs)
apply (simp add: is-alph-def)
apply (simp add: is-alph-def)
apply (drule-tac x=list in spec)
apply (case-tac a)
apply simp
apply simp
apply (rule-tac x=list in exI)
apply simp
done

```

lemma *add-leaves-bdd-all-eq*:
 $list\text{-}all\ P\ (add\text{-}leaves\ tr\ xs) \longleftrightarrow bdd\text{-}all\ P\ tr \wedge list\text{-}all\ P\ xs$
by (induct tr arbitrary: xs) (auto simp add: list-all-iff)

lemmas *add-leaves-bdd-all-eq'* =
add-leaves-bdd-all-eq [**where** xs=[], *simplified*, *symmetric*]

lemma *add-leaves-mono*:
 $set\ xs \subseteq set\ ys \implies set\ (add\text{-}leaves\ tr\ xs) \subseteq set\ (add\text{-}leaves\ tr\ ys)$
by (induct tr arbitrary: xs ys) auto

lemma *add-leaves-binop-subset*:
 $set\ (add\text{-}leaves\ (bdd\text{-}binop\ f\ b\ b')\ [f\ x\ y.\ x \leftarrow xs,\ y \leftarrow ys]) \subseteq$
 $(\bigcup x \in set\ (add\text{-}leaves\ b\ xs).\ \bigcup y \in set\ (add\text{-}leaves\ b'\ y).\ \{f\ x\ y\})\ (is\ ?A \subseteq ?B)$

proof –

have $?A \subseteq (\bigcup x \in set\ (add\text{-}leaves\ b\ xs).\ f\ x\ 'set\ (add\text{-}leaves\ b'\ ys))$

proof (induct f b b' arbitrary: xs ys rule: bdd-binop.induct)

case (1 f x y xs ys) **then show** ?case **by** auto

next

case (2 f l r y xs ys) **then show** ?case

apply auto

apply (drule-tac ys1=[f x y. x ← add-leaves l xs, y ← List.insert y ys] **in**
 rev-subsetD [OF - add-leaves-mono])

apply auto

apply (drule meta-spec, drule meta-spec, drule subsetD, assumption)

apply simp

done

next

case (3 f x l r xs ys) **then show** ?case

apply auto

apply (drule-tac ys1=[f x y. x ← List.insert x xs, y ← add-leaves l ys] **in**
 rev-subsetD [OF - add-leaves-mono])

```

apply auto
apply (drule meta-spec, drule meta-spec, drule subsetD, assumption)
apply simp
done
next
case ( $4 f l_1 r_1 l_2 r_2 xs ys$ ) then show ?case
apply auto
apply (drule-tac ys1=[f x y. x ← add-leaves l1 xs, y ← add-leaves l2 ys] in
  rev-subsetD [OF - add-leaves-mono])
apply simp
apply (drule meta-spec, drule meta-spec, drule subsetD, assumption)
apply simp
done
qed
also have ( $\bigcup x \in \text{set } (add-leaves b xs). f x \text{ ' set } (add-leaves b' ys) = ?B$ )
  by auto
finally show ?thesis .
qed

```

3 DFAs

```

type-synonym bddtable = nat bdd list
type-synonym astate = bool list
type-synonym dfa = bddtable × astate

```

definition

```

dfa-is-node :: dfa ⇒ nat ⇒ bool where
dfa-is-node A = ( $\lambda q. q < \text{length } (fst A)$ )

```

definition

```

wf-dfa :: dfa ⇒ nat ⇒ bool where
wf-dfa A n =
  (list-all (bddh n) (fst A) ∧
   list-all (bdd-all (dfa-is-node A)) (fst A) ∧
   length (snd A) = length (fst A) ∧
   length (fst A) > 0)

```

definition

```

dfa-trans :: dfa ⇒ nat ⇒ bool list ⇒ nat where
dfa-trans A q bs ≡ bdd-lookup (fst A ! q) bs

```

definition

```

dfa-accepting :: dfa ⇒ nat ⇒ bool where
dfa-accepting A q = snd A ! q

```

```

locale aut-dfa =

```

```

  fixes A n
  assumes well-formed: wf-dfa A n

```

```

sublocale aut-dfa < Automaton dfa-trans A dfa-is-node A is-alph n

```

proof

fix $q\ a$
assume Q : $\text{dfa-is-node } A\ q$ **and** A : $\text{is-alfh } n\ a$
hence QL : $q < \text{length } (\text{fst } A)$ **by** ($\text{simp add: dfa-is-node-def}$)
with $\text{well-formed } A$ **have** H : $\text{bddh } (\text{length } a)\ (\text{fst } A\ !\ q)$ **by** ($\text{simp add: wf-dfa-def list-all-iff is-alfh-def}$)
from QL well-formed **have** $\text{bdd-all } (\text{dfa-is-node } A)\ (\text{fst } A\ !\ q)$ **by** ($\text{simp add: wf-dfa-def list-all-iff}$)
with H **show** $\text{dfa-is-node } A\ (\text{dfa-trans } A\ q\ a)$ **by** ($\text{simp add: dfa-trans-def bdd-all-bdd-lookup}$)
qed

context aut-dfa **begin**

lemmas $\text{trans-is-node} = \text{trans-is-node}$

lemmas $\text{steps-is-node} = \text{steps-is-node}$

lemmas $\text{reach-is-node} = \text{reach-is-node}$

end

lemmas $\text{dfa-trans-is-node} = \text{aut-dfa.trans-is-node}$ [$OF\ \text{aut-dfa.intro}$]

lemmas $\text{dfa-steps-is-node} = \text{aut-dfa.steps-is-node}$ [$OF\ \text{aut-dfa.intro}$]

lemmas $\text{dfa-reach-is-node} = \text{aut-dfa.reach-is-node}$ [$OF\ \text{aut-dfa.intro}$]

abbreviation $\text{dfa-steps } A \equiv \text{foldl } (\text{dfa-trans } A)$

abbreviation $\text{dfa-accepts } A \equiv \text{accepts } (\text{dfa-trans } A)\ (\text{dfa-accepting } A)\ 0$

abbreviation $\text{dfa-reach } A \equiv \text{reach } (\text{dfa-trans } A)$

lemma $\text{dfa-startnode-is-node: wf-dfa } A\ n \implies \text{dfa-is-node } A\ 0$

by ($\text{simp add: dfa-is-node-def wf-dfa-def}$)

3.1 Minimization

primrec $\text{make-tr} :: (\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a\ \text{list}$

where

$\text{make-tr } f\ 0\ i = []$

| $\text{make-tr } f\ (\text{Suc } n)\ i = f\ i \# \text{make-tr } f\ n\ (\text{Suc } i)$

primrec $\text{fold-map-idx} :: (\text{nat} \Rightarrow 'c \Rightarrow 'a \Rightarrow 'c \times 'b) \Rightarrow \text{nat} \Rightarrow 'c \Rightarrow 'a\ \text{list} \Rightarrow 'c \times 'b\ \text{list}$

where

$\text{fold-map-idx } f\ i\ y\ [] = (y, [])$

| $\text{fold-map-idx } f\ i\ y\ (x \# xs) =$

$(\text{let } (y', x') = f\ i\ y\ x\ \text{in}$

$\text{let } (y'', xs') = \text{fold-map-idx } f\ (\text{Suc } i)\ y'\ xs\ \text{in } (y'', x' \# xs'))$

definition $\text{init-tr} :: \text{dfa} \Rightarrow \text{bool list list}$ **where**

$\text{init-tr} = (\lambda(\text{bd}, \text{as}). \text{make-tr } (\lambda i. \text{make-tr } (\lambda j. \text{as } !\ i \neq \text{as } !\ j)\ i)\ 0)\ (\text{length } \text{bd} - 1)\ 1)$

definition $\text{tr-lookup} :: \text{bool list list} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$tr\text{-lookup} = (\lambda T i j. (if\ i = j\ then\ False\ else\ if\ i > j\ then\ T\ !\ (i - 1)\ !\ j\ else\ T\ !\ (j - 1)\ !\ i))$

fun $check\text{-}eq :: nat\ bdd \Rightarrow nat\ bdd \Rightarrow bool\ list\ list \Rightarrow bool$ **where**
 $check\text{-}eq\ (Leaf\ i)\ (Leaf\ j)\ T = (\neg\ tr\text{-}lookup\ T\ i\ j) \mid$
 $check\text{-}eq\ (Branch\ l\ r)\ (Leaf\ i)\ T = (check\text{-}eq\ l\ (Leaf\ i)\ T \wedge check\text{-}eq\ r\ (Leaf\ i)\ T) \mid$
 $check\text{-}eq\ (Leaf\ i)\ (Branch\ l\ r)\ T = (check\text{-}eq\ (Leaf\ i)\ l\ T \wedge check\text{-}eq\ (Leaf\ i)\ r\ T) \mid$
 $check\text{-}eq\ (Branch\ l1\ r1)\ (Branch\ l2\ r2)\ T = (check\text{-}eq\ l1\ l2\ T \wedge check\text{-}eq\ r1\ r2\ T)$

definition $iter :: dfa \Rightarrow bool\ list\ list \Rightarrow bool \times bool\ list\ list$ **where**
 $iter = (\lambda(bd,as)\ T. fold\text{-}map\text{-}idx\ (\lambda i. fold\text{-}map\text{-}idx\ (\lambda j\ c\ b. let\ b' = b \vee \neg\ check\text{-}eq\ (bd\ !\ i)\ (bd\ !\ j)\ T in\ (c \vee b \neq b', b'))\ 0)\ 1\ False\ T)$

definition $count\text{-}tr :: bool\ list\ list \Rightarrow nat$ **where**
 $count\text{-}tr = foldl\ (foldl\ (\lambda y\ x. if\ x\ then\ y\ else\ Suc\ y))\ 0$

lemma $fold\text{-}map\text{-}idx\text{-}fst\text{-}snd\text{-}eq$:
assumes $f: \bigwedge i\ c\ x. fst\ (f\ i\ c\ x) = (c \vee x \neq snd\ (f\ i\ c\ x))$
shows $fst\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs) = (c \vee xs \neq snd\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs))$
by (*induct xs arbitrary: i c*) (*simp-all add: split-beta f*)

lemma $foldl\text{-}mono$:
assumes $f: \bigwedge x\ y\ y'. y < y' \Longrightarrow f\ y\ x < f\ y'\ x$ **and** $y: y < y'$
shows $foldl\ f\ y\ xs < foldl\ f\ y'\ xs$ **using** y
by (*induct xs arbitrary: y y'*) (*simp-all add: f*)

lemma $fold\text{-}map\text{-}idx\text{-}count$:
assumes $f: \bigwedge i\ c\ x\ y. fst\ (f\ i\ c\ x) = (c \vee g\ y\ (snd\ (f\ i\ c\ x)) < (g\ y\ x::nat))$
and $f': \bigwedge i\ c\ x\ y. g\ y\ (snd\ (f\ i\ c\ x)) \leq g\ y\ x$
and $g: \bigwedge x\ y\ y'. y < y' \Longrightarrow g\ y\ x < g\ y'\ x$
shows $fst\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs) = (c \vee foldl\ g\ y\ (snd\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs)) < foldl\ g\ y\ xs)$
and $foldl\ g\ y\ (snd\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs)) \leq foldl\ g\ y\ xs$
proof (*induct xs arbitrary: i c y*)
case ($Cons\ x\ xs$) {
case 1
show *?case using f' [of y i c x, simplified le-eq-less-or-eq]*
by (*auto simp add: split-beta Cons(1) [of - - g y (snd (f i c x))] f [of - - - y]*)
intro: less-le-trans foldl-mono g Cons
next
case 2
show *?case using f' [of y i c x, simplified le-eq-less-or-eq]*
by (*auto simp add: split-beta intro: order-trans less-imp-le intro!: foldl-mono g Cons*) }
qed *simp-all*

lemma *iter-count*:

assumes *eq*: $(b, T') = \text{iter } (bd, as) T$

and *b*: b

shows $\text{count-tr } T' < \text{count-tr } T$

proof –

let $?f = \text{fold-map-idx } (\lambda i. \text{fold-map-idx } (\lambda j c b.$

$\text{let } b' = b \vee \neg \text{check-eq } (bd ! i) (bd ! j) T$

$\text{in } (c \vee b \neq b', b')) 0) (\text{Suc } 0) \text{False } T$

from *eq* [*symmetric*] *b* **have** *fst* $?f$

by (*auto simp add: iter-def*)

also have $\text{fst } ?f = (\text{False} \vee \text{count-tr } (\text{snd } ?f) < \text{count-tr } T)$

unfolding *count-tr-def*

by (*rule fold-map-idx-count foldl-mono | simp*) $+$

finally show *?thesis*

by (*simp add: eq [THEN arg-cong, of snd, simplified] iter-def*)

qed

function *fixpt* :: $\text{dfa} \Rightarrow \text{bool list list} \Rightarrow \text{bool list list}$ **where**

$\text{fixpt } M T = (\text{let } (b, T2) = \text{iter } M T \text{ in if } b \text{ then } \text{fixpt } M T2 \text{ else } T2)$

by *auto*

termination by (*relation measure* $(\lambda(M, T). \text{count-tr } T)$) (*auto simp: iter-count*)

lemma *fixpt-True[simp]*: $\text{fst } (\text{iter } M T) \Longrightarrow \text{fixpt } M T = \text{fixpt } M (\text{snd } (\text{iter } M T))$

by (*simp add: split-beta*)

lemma *fixpt-False[simp]*: $\neg (\text{fst } (\text{iter } M T)) \Longrightarrow \text{fixpt } M T = T$

by (*simp add: split-beta iter-def fold-map-idx-fst-snd-eq*)

declare *fixpt.simps* [*simp del*]

lemma *fixpt-induct*:

assumes *H*: $\bigwedge M T. (\text{fst } (\text{iter } M T) \Longrightarrow P M (\text{snd } (\text{iter } M T))) \Longrightarrow P M T$

shows $P M T$

proof (*induct M T rule: fixpt.induct*)

case (1 *M T*)

show *?case* **by** (*rule H*) (*rule 1 [OF refl prod.collapse]*)

qed

definition *dist-nodes* :: $\text{dfa} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**

$\text{dist-nodes} = (\lambda M n m p q. \exists w. \text{length } w = n \wedge \text{list-all } (\text{is-alph } m) w \wedge$

$\text{dfa-accepting } M (\text{dfa-steps } M p w) \neq \text{dfa-accepting } M (\text{dfa-steps } M q w))$

definition *wf-tr* :: $\text{dfa} \Rightarrow \text{bool list list} \Rightarrow \text{bool}$ **where**

$\text{wf-tr} = (\lambda M T. \text{length } T = \text{length } (\text{fst } M) - 1 \wedge (\forall i < \text{length } T. \text{length } (T ! i) = i + 1))$

lemma *make-tr-len*: $\text{length } (\text{make-tr } f n i) = n$

by (*induct n arbitrary: i*) *simp-all*

lemma *make-tr-nth*: $j < n \implies \text{make-tr } f \ n \ i \ ! \ j = f \ (i + j)$

by (*induct* n *arbitrary*: $i \ j$) (*auto simp add*: *nth-Cons'*)

lemma *init-tr-wf*: $\text{wf-tr } M \ (\text{init-tr } M)$

by (*simp add*: *init-tr-def wf-tr-def split-beta make-tr-len make-tr-nth*)

lemma *fold-map-idx-len*: $\text{length} \ (\text{snd} \ (\text{fold-map-idx } f \ i \ y \ xs)) = \text{length} \ xs$

by (*induct* xs *arbitrary*: $i \ y$) (*simp-all add*: *split-beta*)

lemma *fold-map-idx-nth*: $j < \text{length} \ xs \implies$

$\text{snd} \ (\text{fold-map-idx } f \ i \ y \ xs) \ ! \ j = \text{snd} \ (f \ (i + j) \ (\text{fst} \ (\text{fold-map-idx } f \ i \ y \ (\text{take } j \ xs)))) \ (xs \ ! \ j))$

by (*induct* xs *arbitrary*: $i \ j \ y$) (*simp-all add*: *split-beta nth-Cons' take-Cons'*)

lemma *init-tr-dist-nodes*:

assumes *dfa-is-node* $M \ q$ **and** $p < q$

shows $\text{tr-lookup} \ (\text{init-tr } M) \ q \ p = \text{dist-nodes} \ M \ 0 \ v \ p \ q$

proof –

have 1: $\text{dist-nodes} \ M \ 0 \ v \ p \ q = (\text{snd} \ M \ ! \ p \neq \text{snd} \ M \ ! \ q)$ **by** (*simp add*: *dist-nodes-def dfa-accepting-def*)

from *assms* **have** $\text{tr-lookup} \ (\text{init-tr } M) \ q \ p = (\text{snd} \ M \ ! \ p \neq \text{snd} \ M \ ! \ q)$

by (*auto simp add*: *dfa-is-node-def init-tr-def tr-lookup-def make-tr-nth split-beta*)

with 1 **show** *?thesis* **by** *simp*

qed

lemma *dist-nodes-suc*:

$\text{dist-nodes} \ M \ (\text{Suc } n) \ v \ p \ q = (\exists \ bs. \text{is-alph } v \ bs \wedge \text{dist-nodes} \ M \ n \ v \ (\text{dfa-trans } M \ p \ bs) \ (\text{dfa-trans } M \ q \ bs))$

proof

assume $\text{dist-nodes} \ M \ (\text{Suc } n) \ v \ p \ q$

then obtain w **where** W : $\text{length } w = \text{Suc } n$ **and** L : *list-all* $(\text{is-alph } v) \ w$ **and** A : $\text{dfa-accepting } M \ (\text{dfa-steps } M \ p \ w) \neq \text{dfa-accepting } M \ (\text{dfa-steps } M \ q \ w)$ **unfolding** *dist-nodes-def* **by** *blast*

then obtain $b \ bs$ **where** B : $w = b \ \# \ bs$ **by** (*cases* w) *auto*

from A **have** $A2$: $\text{dfa-accepting } M \ (\text{dfa-steps } M \ (\text{dfa-trans } M \ p \ b) \ bs) \neq \text{dfa-accepting } M \ (\text{dfa-steps } M \ (\text{dfa-trans } M \ q \ b) \ bs)$

unfolding B **by** *simp*

with $W \ B \ L$ **show** $\exists \ bs. \text{is-alph } v \ bs \wedge \text{dist-nodes} \ M \ n \ v \ (\text{dfa-trans } M \ p \ bs) \ (\text{dfa-trans } M \ q \ bs)$ **by** (*auto simp*: *dist-nodes-def*)

next

assume $\exists \ bs. \text{is-alph } v \ bs \wedge \text{dist-nodes} \ M \ n \ v \ (\text{dfa-trans } M \ p \ bs) \ (\text{dfa-trans } M \ q \ bs)$

then obtain $b \ bs$ **where** W : $\text{length } bs = n$ **and** V : *is-alph* $v \ b$ **and** V' : *list-all* $(\text{is-alph } v) \ bs$

and A : $\text{dfa-accepting } M \ (\text{dfa-steps } M \ (\text{dfa-trans } M \ p \ b) \ bs) \neq \text{dfa-accepting } M \ (\text{dfa-steps } M \ (\text{dfa-trans } M \ q \ b) \ bs)$

unfolding *dist-nodes-def* **by** *blast*

hence $\text{dfa-accepting } M \ (\text{dfa-steps } M \ p \ (b \ \# \ bs)) \neq \text{dfa-accepting } M \ (\text{dfa-steps } M$

$q (b \# bs)$ **by** *simp*
moreover from W **have** $\text{length } (b \# bs) = \text{Suc } n$ **by** *simp*
moreover from $V V'$ **have** $\text{list-all } (\text{is-alph } v) (b \# bs)$ **by** *simp*
ultimately show $\text{dist-nodes } M (\text{Suc } n) v p q$ **unfolding** dist-nodes-def **by** *blast*
qed

lemma *bdd-lookup-append*:
assumes $\text{bddh } n B$ **and** $\text{length } bs \geq n$
shows $\text{bdd-lookup } B (bs @ w) = \text{bdd-lookup } B bs$
using *assms*
proof (*induct* $B bs$ *arbitrary: n rule: bdd-lookup.induct*)
case ($2 l r b bs n$)
then obtain n' **where** $N: n = \text{Suc } n'$ **by** (*cases n*) *simp+*
with 2 show $?case$ **by** (*cases b*) *auto*
qed *simp+*

lemma *bddh-exists*: $\exists n. \text{bddh } n B$
proof (*induct* B)
case (*Branch l r*)
then obtain $n m$ **where** $L: \text{bddh } n l$ **and** $R: \text{bddh } m r$ **by** *blast*
with $\text{bddh-ge}[of n \text{max } n m l]$ $\text{bddh-ge}[of m \text{max } n m r]$ **have** $\text{bddh } (\text{Suc } (\text{max } n m))$ (*Branch l r*) **by** *simp*
thus $?case$ **by** (*rule exI*)
qed *simp*

lemma *check-eq-dist-nodes*:
assumes $\forall p q. \text{dfa-is-node } M q \wedge p < q \longrightarrow \text{tr-lookup } T q p = (\exists n < m. \text{dist-nodes } M n v p q)$ **and** $m > 0$
and $\text{bdd-all } (\text{dfa-is-node } M) l$ **and** $\text{bdd-all } (\text{dfa-is-node } M) r$
shows $(\neg \text{check-eq } l r T) = (\exists bs. \text{bddh } (\text{length } bs) l \wedge \text{bddh } (\text{length } bs) r \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } l bs) (\text{bdd-lookup } r bs)))$
using *assms* **proof** (*induct* $l r T$ *rule: check-eq.induct*)
case ($1 i j T$)
have $i < j \vee i = j \vee i > j$ **by** *auto*
thus $?case$ **by** (*elim disjE*) (*insert 1, auto simp: dist-nodes-def tr-lookup-def*)
next
case ($2 l r i T$)
hence *IV1*: $(\neg \text{check-eq } l (\text{Leaf } i) T) = (\exists bs. \text{bddh } (\text{length } bs) l \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } l bs) (\text{bdd-lookup } (\text{Leaf } i) bs)))$
by *simp*
from 2 have *IV2*: $(\neg \text{check-eq } r (\text{Leaf } i) T) = (\exists bs. \text{bddh } (\text{length } bs) r \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } r bs) (\text{bdd-lookup } (\text{Leaf } i) bs)))$
by *simp*
have $(\neg \text{check-eq } (\text{Branch } l r) (\text{Leaf } i) T) = (\neg \text{check-eq } l (\text{Leaf } i) T \vee \neg \text{check-eq } r (\text{Leaf } i) T)$ **by** *simp*
also have $\dots = (\exists bs. \text{bddh } (\text{length } bs) (\text{Branch } l r) \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Branch } l r) bs) (\text{bdd-lookup } (\text{Leaf } i) bs)))$
(is $(?L \vee ?R) = ?E$ **)**
proof

assume $?L \vee ?R$
thus $?E$ **proof** (*elim disjE*)
assume $?L$
then obtain bs **where** $O: bddh (length\ bs)\ l \wedge bddh (length\ bs)\ (Leaf\ i) \wedge$
 $(\exists n < m. dist-nodes\ M\ n\ v\ (bdd-lookup\ l\ bs)\ (bdd-lookup\ (Leaf\ i)\ bs))$ **unfolding**
IV1 by blast
from *bddh-exists* **obtain** k **where** $B: bddh\ k\ r$ **by blast**
with O **have** $bddh (length\ bs + k)\ r$ **and** $bddh (length\ bs + k)\ l$ **and** $bddh$
 $(length\ bs + k)\ (Leaf\ i)$ **by** (*simp add: bddh-ge[of k length bs + k] bddh-ge[of length*
 $bs\ length\ bs + k]$)**+**
with O **have** $bddh (length\ (False\ \#\ bs\ @\ replicate\ k\ False))\ (Branch\ l\ r) \wedge$
 $bddh (length\ (False\ \#\ bs\ @\ replicate\ k\ False))\ (Leaf\ i) \wedge (\exists n < m. dist-nodes\ M\ n$
 $v\ (bdd-lookup\ (Branch\ l\ r)\ (False\ \#\ bs\ @\ replicate\ k\ False))\ (bdd-lookup\ (Leaf\ i)$
 $(False\ \#\ bs\ @\ replicate\ k\ False)))$ **by** (*auto simp: bdd-lookup-append*)
thus $?thesis$ **by** (*rule exI*)
next
assume $?R$
then obtain bs **where** $O: bddh (length\ bs)\ r \wedge bddh (length\ bs)\ (Leaf\ i) \wedge$
 $(\exists n < m. dist-nodes\ M\ n\ v\ (bdd-lookup\ r\ bs)\ (bdd-lookup\ (Leaf\ i)\ bs))$ **unfolding**
IV2 by blast
from *bddh-exists* **obtain** k **where** $B: bddh\ k\ l$ **by blast**
with O **have** $bddh (length\ bs + k)\ l$ **and** $bddh (length\ bs + k)\ r$ **and** $bddh$
 $(length\ bs + k)\ (Leaf\ i)$ **by** (*simp add: bddh-ge[of k length bs + k] bddh-ge[of length*
 $bs\ length\ bs + k]$)**+**
with O **have** $bddh (length\ (True\ \#\ bs\ @\ replicate\ k\ False))\ (Branch\ l\ r) \wedge$
 $bddh (length\ (True\ \#\ bs\ @\ replicate\ k\ False))\ (Leaf\ i) \wedge (\exists n < m. dist-nodes\ M\ n$
 $v\ (bdd-lookup\ (Branch\ l\ r)\ (True\ \#\ bs\ @\ replicate\ k\ False))\ (bdd-lookup\ (Leaf\ i)$
 $(True\ \#\ bs\ @\ replicate\ k\ False)))$ **by** (*auto simp: bdd-lookup-append*)
thus $?thesis$ **by** (*rule exI*)
qed
next
assume $?E$
then obtain bs **where** $O: bddh (length\ bs)\ (Branch\ l\ r) \wedge bddh (length\ bs)$
 $(Leaf\ i) \wedge (\exists n < m. dist-nodes\ M\ n\ v\ (bdd-lookup\ (Branch\ l\ r)\ bs)\ (bdd-lookup\ (Leaf$
 $i)\ bs))$ **by blast**
then obtain $b\ br$ **where** $B: bs = b\ \#\ br$ **by** (*cases bs*) *auto*
with O **IV1 IV2 show** $?L \vee ?R$ **by** (*cases b*) *auto*
qed
finally show $?case$ **by** *simp*
next
case $(3\ i\ l\ r\ T)$
hence $IV1: (\neg\ check-eq\ (Leaf\ i)\ l\ T) = (\exists\ bs. bddh (length\ bs)\ l \wedge bddh (length$
 $bs)\ (Leaf\ i) \wedge (\exists n < m. dist-nodes\ M\ n\ v\ (bdd-lookup\ (Leaf\ i)\ bs)\ (bdd-lookup\ l\ bs)))$
by *simp*
from 3 **have** $IV2: (\neg\ check-eq\ (Leaf\ i)\ r\ T) = (\exists\ bs. bddh (length\ bs)\ r \wedge bddh$
 $(length\ bs)\ (Leaf\ i) \wedge (\exists n < m. dist-nodes\ M\ n\ v\ (bdd-lookup\ (Leaf\ i)\ bs)\ (bdd-lookup$
 $r\ bs)))$ **by** *simp*
have $(\neg\ check-eq\ (Leaf\ i)\ (Branch\ l\ r)\ T) = (\neg\ check-eq\ (Leaf\ i)\ l\ T \vee \neg\ check-eq$
 $(Leaf\ i)\ r\ T)$ **by** *simp*

also have $\dots = (\exists bs. \text{bddh } (\text{length } bs) (\text{Branch } l r) \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) bs) (\text{bdd-lookup } (\text{Branch } l r) bs)))$
(is $(?L \vee ?R) = ?E$)

proof

assume $?L \vee ?R$

thus $?E$ **proof** (*elim disjE*)

assume $?L$

then obtain bs **where** $O: \text{bddh } (\text{length } bs) l \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) bs) (\text{bdd-lookup } l bs))$ **unfolding** *IV1* **by** *blast*

from *bddh-exists* **obtain** k **where** $B: \text{bddh } k r$ **by** *blast*

with O **have** $\text{bddh } (\text{length } bs + k) r$ **and** $\text{bddh } (\text{length } bs + k) l$ **and** $\text{bddh } (\text{length } bs + k) (\text{Leaf } i)$ **by** (*simp add: bddh-ge[of k length bs + k] bddh-ge[of length bs length bs + k]*)**+**

with O **have** $\text{bddh } (\text{length } (\text{False} \# bs @ \text{replicate } k \text{ False})) (\text{Branch } l r) \wedge \text{bddh } (\text{length } (\text{False} \# bs @ \text{replicate } k \text{ False})) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) (\text{False} \# bs @ \text{replicate } k \text{ False})) (\text{bdd-lookup } (\text{Branch } l r) (\text{False} \# bs @ \text{replicate } k \text{ False})))$ **by** (*auto simp: bdd-lookup-append*)

thus $?thesis$ **by** (*rule exI*)

next

assume $?R$

then obtain bs **where** $O: \text{bddh } (\text{length } bs) r \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) bs) (\text{bdd-lookup } r bs))$ **unfolding** *IV2* **by** *blast*

from *bddh-exists* **obtain** k **where** $B: \text{bddh } k l$ **by** *blast*

with O **have** $\text{bddh } (\text{length } bs + k) l$ **and** $\text{bddh } (\text{length } bs + k) r$ **and** $\text{bddh } (\text{length } bs + k) (\text{Leaf } i)$ **by** (*simp add: bddh-ge[of k length bs + k] bddh-ge[of length bs length bs + k]*)**+**

with O **have** $\text{bddh } (\text{length } (\text{True} \# bs @ \text{replicate } k \text{ False})) (\text{Branch } l r) \wedge \text{bddh } (\text{length } (\text{True} \# bs @ \text{replicate } k \text{ False})) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) (\text{True} \# bs @ \text{replicate } k \text{ False})) (\text{bdd-lookup } (\text{Branch } l r) (\text{True} \# bs @ \text{replicate } k \text{ False})))$ **by** (*auto simp: bdd-lookup-append*)

thus $?thesis$ **by** (*rule exI*)

qed

next

assume $?E$

then obtain bs **where** $O: \text{bddh } (\text{length } bs) (\text{Branch } l r) \wedge \text{bddh } (\text{length } bs) (\text{Leaf } i) \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } (\text{Leaf } i) bs) (\text{bdd-lookup } (\text{Branch } l r) bs))$ **by** *blast*

then obtain b br **where** $B: bs = b \# br$ **by** (*cases bs*) *auto*

with O *IV1* *IV2* **show** $?L \vee ?R$ **by** (*cases b*) *auto*

qed

finally show $?case$ **by** *simp*

next

case $(4 l1 r1 l2 r2 T)$

hence *IV1*: $(\neg \text{check-eq } l1 l2 T) = (\exists bs. \text{bddh } (\text{length } bs) l1 \wedge \text{bddh } (\text{length } bs) l2 \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } l1 bs) (\text{bdd-lookup } l2 bs)))$ **by** *simp*

from 4 **have** *IV2*: $(\neg \text{check-eq } r1 r2 T) = (\exists bs. \text{bddh } (\text{length } bs) r1 \wedge \text{bddh } (\text{length } bs) r2 \wedge (\exists n < m. \text{dist-nodes } M n v (\text{bdd-lookup } r1 bs) (\text{bdd-lookup } r2 bs)))$

by simp
have $(\neg \text{check-eq } (\text{Branch } l1 \ r1) \ (\text{Branch } l2 \ r2) \ T) = (\neg \text{check-eq } l1 \ l2 \ T \vee \neg \text{check-eq } r1 \ r2 \ T)$ **by simp**
also have $\dots = (\exists bs. \text{bddh } (\text{length } bs) \ (\text{Branch } l1 \ r1) \wedge \text{bddh } (\text{length } bs) \ (\text{Branch } l2 \ r2) \wedge (\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } (\text{Branch } l1 \ r1) \ bs) \ (\text{bdd-lookup } (\text{Branch } l2 \ r2) \ bs)))$
(is $(?L \vee ?R) = (\exists bs. ?E \ bs)$ **) proof**
assume $?L \vee ?R$
thus $\exists bs. ?E \ bs$ **proof** *(elim disjE)*
assume $?L$
then obtain bs **where** $O: \text{bddh } (\text{length } bs) \ l1 \wedge \text{bddh } (\text{length } bs) \ l2 \wedge (\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } l1 \ bs) \ (\text{bdd-lookup } l2 \ bs))$ **unfolding** $IV1$ **by** *blast*
from *bddh-exists* **obtain** $k1 \ k2$ **where** $K1: \text{bddh } k1 \ r1$ **and** $K2: \text{bddh } k2 \ r2$
by *blast*
with O **have** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ l1$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ l2$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ r1$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ r2$
by *(simp add: bddh-ge[of length bs length bs + max k1 k2] bddh-ge[of k1 length bs + max k1 k2] bddh-ge[of k2 length bs + max k1 k2])*
with O **have** $\text{bddh } (\text{length } (\text{False} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{Branch } l1 \ r1) \wedge \text{bddh } (\text{length } (\text{False} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{Branch } l2 \ r2)$
 \wedge
 $(\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } (\text{Branch } l1 \ r1) \ (\text{False} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{bdd-lookup } (\text{Branch } l2 \ r2) \ (\text{False} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})))$
by *(auto simp: bdd-lookup-append)*
thus $?thesis$ **by** *(rule exI)*
next
assume $?R$
then obtain bs **where** $O: \text{bddh } (\text{length } bs) \ r1 \wedge \text{bddh } (\text{length } bs) \ r2 \wedge (\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } r1 \ bs) \ (\text{bdd-lookup } r2 \ bs))$ **unfolding** $IV2$ **by** *blast*
from *bddh-exists* **obtain** $k1 \ k2$ **where** $K1: \text{bddh } k1 \ l1$ **and** $K2: \text{bddh } k2 \ l2$
by *blast*
with O **have** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ l1$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ l2$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ r1$ **and** $\text{bddh } (\text{length } bs + \max k1 \ k2) \ r2$
by *(simp add: bddh-ge[of length bs length bs + max k1 k2] bddh-ge[of k1 length bs + max k1 k2] bddh-ge[of k2 length bs + max k1 k2])*
with O **have** $\text{bddh } (\text{length } (\text{True} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{Branch } l1 \ r1) \wedge \text{bddh } (\text{length } (\text{True} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{Branch } l2 \ r2)$
 \wedge
 $(\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } (\text{Branch } l1 \ r1) \ (\text{True} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})) \ (\text{bdd-lookup } (\text{Branch } l2 \ r2) \ (\text{True} \# \text{bs} \ @ \ \text{replicate } (\max k1 \ k2) \ \text{False})))$
by *(auto simp: bdd-lookup-append)*
thus $?thesis$ **by** *(rule exI)*
qed
next
assume $\exists bs. ?E \ bs$
then obtain bs **where** $O: ?E \ bs$ **by** *blast*
then obtain $b \ br$ **where** $B: bs = b \# \ br$ **by** *(cases bs) auto*

with O $IV1$ $IV2$ **show** $?L \vee ?R$ **by** (*cases b*) *auto*
qed
finally show $?case$ **by** *simp*
qed

lemma *iter-wf*: $wf\text{-tr } M \ T \implies wf\text{-tr } M \ (snd \ (iter \ M \ T))$
by (*simp add: wf-tr-def iter-def fold-map-idx-len fold-map-idx-nth split-beta*)

lemma *fixpt-wf*: $wf\text{-tr } M \ T \implies wf\text{-tr } M \ (fixpt \ M \ T)$
proof (*induct M T rule: fixpt-induct*)
case ($1 \ M \ T$)
show $?case$ **proof** (*cases fst (iter M T)*)
case *True* **with** 1 **show** $?thesis$ **by** (*simp add: iter-wf*)
next
case *False* **with** 1 **show** $?thesis$ **by** *simp*
qed
qed

lemma *list-split*:
assumes $n \leq length \ bss$
shows $\exists b \ bs. \ bss = b \ @ \ bs \wedge length \ b = n$
using *assms* **proof** (*induct bss arbitrary: n*)
case (*Cons a as*)
show $?case$ **proof** (*cases n*)
case (*Suc n'*)
with *Cons* **have** $\exists b \ bs. \ as = b \ @ \ bs \wedge length \ b = n'$ **by** *simp*
then obtain $b \ bs$ **where** $B: as = b \ @ \ bs \wedge length \ b = n'$ **by** *blast*
with *Suc Cons* **have** $a \ \# \ as = (a \ \# \ b) \ @ \ bs \wedge length \ (a \ \# \ b) = n$ **by** *simp*
thus $?thesis$ **by** *blast*
qed *simp*
qed *simp*

lemma *iter-dist-nodes*:
assumes $wf\text{-tr } M \ T$
and $wf\text{-dfa } M \ v$
and $\forall p \ q. \ dfa\text{-is-node } M \ q \wedge p < q \implies tr\text{-lookup } T \ q \ p = (\exists n < m. \ dist\text{-nodes } M \ n \ v \ p \ q)$ **and** $m > 0$
and $dfa\text{-is-node } M \ q$ **and** $p < q$
shows $tr\text{-lookup } (snd \ (iter \ M \ T)) \ q \ p = (\exists n < Suc \ m. \ dist\text{-nodes } M \ n \ v \ p \ q)$
proof –
from *assms* **obtain** m' **where** $M': m = Suc \ m'$ **by** (*cases m*) *simp+*
have $C: (\neg \ check\text{-eq } (fst \ M \ ! \ q) \ (fst \ M \ ! \ p) \ T) = (\exists n < m. \ dist\text{-nodes } M \ (Suc \ n) \ v \ p \ q)$ **proof**
assume $\neg \ check\text{-eq } (fst \ M \ ! \ q) \ (fst \ M \ ! \ p) \ T$
with *assms* **have** $\exists bs. \ bddh \ (length \ bs) \ (fst \ M \ ! \ q) \wedge bddh \ (length \ bs) \ (fst \ M \ ! \ p) \wedge (\exists n < m. \ dist\text{-nodes } M \ n \ v \ (bdd\text{-lookup } (fst \ M \ ! \ q) \ bs) \ (bdd\text{-lookup } (fst \ M \ ! \ p) \ bs))$
by (*simp add: check-eq-dist-nodes wf-dfa-def list-all-iff dfa-is-node-def*)
then obtain $bs \ n \ bss$ **where** $X: bddh \ (length \ bs) \ (fst \ M \ ! \ q) \wedge bddh \ (length \ bs)$

$(fst M ! p) \wedge n < m \wedge$
 $length\ bss = n \wedge list\text{-}all\ (is\text{-}alph\ v)\ bss \wedge dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M$
 $(bdd\text{-}lookup\ (fst\ M ! q)\ bs)\ bss) \neq dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M\ (bdd\text{-}lookup\ (fst$
 $M ! p)\ bs)\ bss)$

unfolding *dist-nodes-def* **by** *blast*

from *list-split*[of $v\ bs$ @ *replicate* $v\ False$] **have** $\exists b' bs'. bs @ replicate\ v\ False$
 $= b' @ bs' \wedge length\ b' = v$ **by** *simp*

then obtain $b' bs'$ **where** $V: bs @ replicate\ v\ False = b' @ bs' \wedge length\ b' =$
 v **by** *blast*

with X *bdd-lookup-append*[of $length\ bs\ fst\ M ! q\ bs$ *replicate* $v\ False$] *bdd-lookup-append*[of
 $length\ bs\ fst\ M ! p\ bs$ *replicate* $v\ False$]

have 1: $dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M\ (bdd\text{-}lookup\ (fst\ M ! q)\ (bs @ replicate\ v$
 $False))\ bss) \neq dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M\ (bdd\text{-}lookup\ (fst\ M ! p)\ (bs @ replicate$
 $v\ False))\ bss)$ **by** *simp*

from *assms* **have** $bddh\ v\ (fst\ M ! q) \wedge bddh\ v\ (fst\ M ! p)$ **by** (*simp add:*
wf-dfa-def dfa-is-node-def list-all-iff)

with 1 V **have** $dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M\ (dfa\text{-}trans\ M\ q\ b')\ bss) \neq$
 $dfa\text{-}accepting\ M\ (dfa\text{-}steps\ M\ (dfa\text{-}trans\ M\ p\ b')\ bss)$ **by** (*auto simp: bdd-lookup-append*
dfa-trans-def)

with $X\ V$ **have** $is\text{-}alph\ v\ b' \wedge dist\text{-}nodes\ M\ n\ v\ (dfa\text{-}trans\ M\ p\ b')\ (dfa\text{-}trans$
 $M\ q\ b')$ **by** (*auto simp: dist-nodes-def is-alph-def*)

hence $dist\text{-}nodes\ M\ (Suc\ n)\ v\ p\ q$ **by** (*auto simp: dist-nodes-suc*)

with X **show** $\exists n < m. dist\text{-}nodes\ M\ (Suc\ n)\ v\ p\ q$ **by** *auto*

next

assume $\exists n < m. dist\text{-}nodes\ M\ (Suc\ n)\ v\ p\ q$

hence $\exists bs. \exists n < m. is\text{-}alph\ v\ bs \wedge dist\text{-}nodes\ M\ n\ v\ (dfa\text{-}trans\ M\ p\ bs)\ (dfa\text{-}trans$
 $M\ q\ bs)$ **by** (*auto simp: dist-nodes-suc*)

then obtain bs **where** $X: \exists n < m. is\text{-}alph\ v\ bs \wedge dist\text{-}nodes\ M\ n\ v\ (dfa\text{-}trans$
 $M\ p\ bs)\ (dfa\text{-}trans\ M\ q\ bs)$ **by** *blast*

hence $BS: length\ bs = v$ **by** (*auto simp: is-alph-def*)

with *assms* **have** $bddh\ (length\ bs)\ (fst\ M ! p) \wedge bddh\ (length\ bs)\ (fst\ M ! q)$
by (*simp add: wf-dfa-def dfa-is-node-def list-all-iff*)

with X **have** $bddh\ (length\ bs)\ (fst\ M ! p) \wedge bddh\ (length\ bs)\ (fst\ M ! q) \wedge$
 $(\exists n < m. dist\text{-}nodes\ M\ n\ v\ (bdd\text{-}lookup\ (fst\ M ! q)\ bs)\ (bdd\text{-}lookup\ (fst\ M ! p)\ bs))$
by (*auto simp: dfa-trans-def dist-nodes-def*)

moreover from *assms* **have** $bdd\text{-}all\ (dfa\text{-}is\text{-}node\ M)\ (fst\ M ! p) \wedge bdd\text{-}all$
 $(dfa\text{-}is\text{-}node\ M)\ (fst\ M ! q)$ **by** (*simp add: wf-dfa-def dfa-is-node-def list-all-iff*)

moreover note *assms*(3,4)

ultimately show $\neg check\text{-}eq\ (fst\ M ! q)\ (fst\ M ! p)\ T$ **by** (*auto simp:*
check-eq-dist-nodes)

qed

from *assms* **have** $tr\text{-}lookup\ (snd\ (iter\ M\ T))\ q\ p =$
 $(if\ tr\text{-}lookup\ T\ q\ p\ then\ True\ else\ \neg\ check\text{-}eq\ (fst\ M ! q)\ (fst\ M ! p)\ T)$

by (*auto simp add: iter-def wf-tr-def split-beta fold-map-idx-nth tr-lookup-def*
dfa-is-node-def)

also have $\dots = (tr\text{-}lookup\ T\ q\ p \vee \neg\ check\text{-}eq\ (fst\ M ! q)\ (fst\ M ! p)\ T)$ **by** *simp*

also from *assms* C **have** $\dots = ((\exists n < m. dist\text{-}nodes\ M\ n\ v\ p\ q) \vee (\exists n < m.$
 $dist\text{-}nodes\ M\ (Suc\ n)\ v\ p\ q))$ **by** *simp*

also have $\dots = (\exists n < m. \text{dist-nodes } M \ n \ v \ p \ q \vee \text{dist-nodes } M \ (\text{Suc } n) \ v \ p \ q)$
by *auto*
also have $\dots = (\exists n < \text{Suc } m. \text{dist-nodes } M \ n \ v \ p \ q)$ **proof**
assume $\exists n < m. \text{dist-nodes } M \ n \ v \ p \ q \vee \text{dist-nodes } M \ (\text{Suc } n) \ v \ p \ q$
then obtain n **where** $D: \text{dist-nodes } M \ n \ v \ p \ q \vee \text{dist-nodes } M \ (\text{Suc } n) \ v \ p \ q$
and $N: n < m$ **by** *blast*
moreover from N **have** $n < \text{Suc } m$ **by** *simp*
ultimately show $\exists n < \text{Suc } m. \text{dist-nodes } M \ n \ v \ p \ q$ **by** (*elim disjE*) *blast+*
next
assume $\exists n < \text{Suc } m. \text{dist-nodes } M \ n \ v \ p \ q$
then obtain n **where** $N: n < \text{Suc } m$ **and** $D: \text{dist-nodes } M \ n \ v \ p \ q$ **by** *blast*
from N **have** $n < m \vee n = m$ **by** *auto*
from this $D \ M'$ **show** $\exists n < m. \text{dist-nodes } M \ n \ v \ p \ q \vee \text{dist-nodes } M \ (\text{Suc } n) \ v \ p \ q$ **by** *auto*
qed
finally show *?thesis* **by** *simp*
qed

lemma *fixpt-dist-nodes'*:

assumes *wf-tr* $M \ T$ **and** *wf-dfa* $M \ v$
and $\forall p \ q. \text{dfa-is-node } M \ q \wedge p < q \longrightarrow \text{tr-lookup } T \ q \ p = (\exists n < m. \text{dist-nodes } M \ n \ v \ p \ q)$ **and** $m > 0$
and *dfa-is-node* $M \ q$ **and** $p < q$
shows $\text{tr-lookup } (\text{fixpt } M \ T) \ q \ p = (\exists n. \text{dist-nodes } M \ n \ v \ p \ q)$
using *assms* **proof** (*induct* $M \ T$ *arbitrary: m rule: fixpt-induct*)
case $(1 \ M \ T \ m)$
let $?T = \text{snd } (\text{iter } M \ T)$
show *?case* **proof** (*cases fst (iter M T)*)
case *True*
{ **fix** $p' \ q'$ **assume** $H: \text{dfa-is-node } M \ q' \wedge p' < q'$
with 1 **have** $\text{tr-lookup } ?T \ q' \ p' = (\exists n < \text{Suc } m. \text{dist-nodes } M \ n \ v \ p' \ q')$ **by**
(*simp only: iter-dist-nodes*)
} **hence** $2: \forall p \ q. \text{dfa-is-node } M \ q \wedge p < q \longrightarrow \text{tr-lookup } ?T \ q \ p = (\exists n < \text{Suc } m. \text{dist-nodes } M \ n \ v \ p \ q)$ **by** *simp* **moreover**
from 1 **have** *wf-tr* $M \ ?T$ **by** (*simp add: iter-wf*) **moreover**
note $1(3,6,7) \ 1(1)[\text{of } \text{Suc } m]$ *True*
ultimately have $\text{tr-lookup } (\text{fixpt } M \ ?T) \ q \ p = (\exists n. \text{dist-nodes } M \ n \ v \ p \ q)$ **by**
simp
with *True* **show** *?thesis* **by** (*simp add: Let-def split-beta*)
next
case *False*
then have $F: \text{snd } (\text{iter } M \ T) = T$ **by** (*simp add: iter-def fold-map-idx-fst-snd-eq split-beta*)
have $C: \bigwedge m'. \forall p \ q. \text{dfa-is-node } M \ q \wedge p < q \longrightarrow \text{tr-lookup } T \ q \ p = (\exists n < m' + m. \text{dist-nodes } M \ n \ v \ p \ q)$
proof –
fix m' **show** $\forall p \ q. \text{dfa-is-node } M \ q \wedge p < q \longrightarrow \text{tr-lookup } T \ q \ p = (\exists n < m' + m. \text{dist-nodes } M \ n \ v \ p \ q)$
proof (*induct m'*)

```

    case 0 with 1 show ?case by simp
  next
  case (Suc m')
  { fix p' q' assume H: dfa-is-node M q' and H2: p' < q'
    note 1(2,3) Suc
    moreover from Suc 1 have 0 < m' + m by simp
    moreover note H H2
    ultimately have tr-lookup (snd (iter M T)) q' p' = (∃ n < Suc (m' +
m). dist-nodes M n v p' q') by (rule iter-dist-nodes)
    with F have tr-lookup T q' p' = (∃ n < Suc m' + m. dist-nodes M n v p'
q') by simp
  } thus ?case by simp
qed
qed
{
  fix p' q' assume H: dfa-is-node M q' ∧ p' < q'
  have tr-lookup T q' p' = (∃ n. dist-nodes M n v p' q') proof
  assume tr-lookup T q' p'
  with H C[of 0] show ∃ n. dist-nodes M n v p' q' by auto
next
  assume H': ∃ n. dist-nodes M n v p' q'
  then obtain n where dist-nodes M n v p' q' by blast
  moreover have n < Suc n + m by simp
  ultimately have ∃ n' < Suc n + m. dist-nodes M n' v p' q' by blast
  with H C[of Suc n] show tr-lookup T q' p' by simp
qed
} hence ∀ p q. dfa-is-node M q ∧ p < q ⟶ tr-lookup T q p = (∃ n. dist-nodes
M n v p q) by simp
with False ⟨dfa-is-node M q⟩ ⟨p < q⟩ show ?thesis by simp
qed
qed

```

lemma *fixpt-dist-nodes*:

```

  assumes wf-dfa M v
  and dfa-is-node M p and dfa-is-node M q
  shows tr-lookup (fixpt M (init-tr M)) p q = (∃ n. dist-nodes M n v p q)
proof –
  { fix p q assume H1: p < q and H2: dfa-is-node M q
    from init-tr-wf have wf-tr M (init-tr M) by simp
    moreover note assms(1)
    moreover {
      fix p' q' assume dfa-is-node M q' and p' < q'
      hence tr-lookup (init-tr M) q' p' = dist-nodes M 0 v p' q' by (rule init-tr-dist-nodes)
      also have ... = (∃ n < 1. dist-nodes M n v p' q') by auto
      finally have tr-lookup (init-tr M) q' p' = (∃ n < 1. dist-nodes M n v p' q') by
simp
    } hence ∀ p q. dfa-is-node M q ∧ p < q ⟶ tr-lookup (init-tr M) q p = (∃ n < 1.
dist-nodes M n v p q) by simp
  moreover note H1 H2

```

ultimately have *tr-lookup* (*fixpt* *M* (*init-tr* *M*)) *q p* = ($\exists n$. *dist-nodes* *M n v p q*) by (*simp only: fixpt-dist-nodes'[of - - - 1]*)
 }
 with *assms*(2,3) **show** *?thesis* by (*auto simp: tr-lookup-def dist-nodes-def*)
qed

primrec *mk-eqcl'* :: *nat option list* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool list list* \Rightarrow *nat option list*

where

mk-eqcl' [] *i j l T* = []
 | *mk-eqcl'* (*x#xs*) *i j l T* = (*if tr-lookup T j i* \vee *x* \neq *None* then *x* else *Some l*) # *mk-eqcl' xs i (Suc j) l T*

lemma *mk-eqcl'-len*: *length (mk-eqcl' xs i j l T)* = *length xs* by (*induct xs arbitrary: j*) *simp+*

function *mk-eqcl* :: *nat option list* \Rightarrow *nat list* \Rightarrow *nat* \Rightarrow *bool list list* \Rightarrow *nat list* \times *nat list* **where**

mk-eqcl [] *zs i T* = ([], *zs*) |
mk-eqcl (*None* # *xs*) *zs i T* = (*let (xs',zs')* = *mk-eqcl (mk-eqcl' xs i (Suc i) (length zs) T) (zs @ [i]) (Suc i) T in (length zs # xs', zs')*) |
mk-eqcl (*Some l* # *xs*) *zs i T* = (*let (xs',zs')* = *mk-eqcl xs zs (Suc i) T in (l # xs', zs')*)

by *pat-completeness auto*

termination by (*lexicographic-order simp: mk-eqcl'-len*)

lemma *mk-eqcl'-bound*:

assumes $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < l$

and $x \in \text{set } (mk-eqcl' xs i j l T)$ **and** $x = \text{Some } k$

shows $k \leq l$

using *assms* **proof** (*induct xs arbitrary: j*)

case (*Cons y xs j*)

hence $x = y \vee x = \text{Some } l \vee x \in \text{set } (mk-eqcl' xs i (Suc j) l T)$ **by** (*cases tr-lookup T j i* \vee $y \neq \text{None}$) *auto*

thus *?case* **proof** (*elim disjE*)

assume $x = y$

hence $x \in \text{set } (y \# xs)$ **by** *simp*

with *Cons*(2)[*of x k*] *Cons*(4) **show** *?thesis* **by** *simp*

qed (*insert Cons, auto*)

qed *simp*

lemma *mk-eqcl'-nth'*:

assumes $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < l$

and $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + j) i \rrbracket \implies xs ! i' = \text{None}$

and $i < j$ **and** $j' < \text{length } xs$

shows $(mk-eqcl' xs i j l T ! j' = \text{Some } l) = (\neg \text{tr-lookup } T (j' + j) i)$

using *assms* **proof** (*induct xs arbitrary: j j'*)

case (*Cons x xs j*)

have $I1: \bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + \text{Suc } j) i \rrbracket \implies xs ! i' = \text{None}$

proof –
fix i' **assume** $H: i' < \text{length } xs \wedge \neg \text{tr-lookup } T (i' + \text{Suc } j) i$
with $\text{Cons}(3)[\text{of } \text{Suc } i']$ **show** $xs ! i' = \text{None}$ **by** *simp*
qed
have $j' = 0 \vee j' > 0$ **by** *auto*
thus $?case$ **proof** (*elim disjE*)
assume $j' > 0$
then obtain j'' **where** $J: j' = \text{Suc } j''$ **by** (*cases j'*) *simp+*
from $\text{Cons}(1)[\text{of } \text{Suc } j j'']$ $I1$ $\text{Cons}(2,4,5)$ J **show** $?thesis$ **by** *simp*
next
assume $H: j' = 0$
with $\text{Cons}(3)[\text{of } 0]$ **have** $\neg \text{tr-lookup } T j i \implies x = \text{None}$ **by** *simp*
with $\text{Cons } H$ **show** $?thesis$ **by** *auto*
qed
qed *simp*

lemma *mk-eqcl'-nth*:
assumes $\bigwedge i' j' k. \llbracket i' < \text{length } xs; j' < \text{length } xs; xs ! i' = \text{Some } k \rrbracket \implies (xs ! j' = \text{Some } k) = (\neg \text{tr-lookup } T (i' + jj) (j' + jj))$
and $\bigwedge a b c. \llbracket a \leq \text{length } T; b \leq \text{length } T; c \leq \text{length } T; \neg \text{tr-lookup } T a b; \neg \text{tr-lookup } T b c \rrbracket \implies \neg \text{tr-lookup } T a c$
and $\text{length } xs + jj = \text{length } T + 1$
and $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < l$
and $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + jj) i \rrbracket \implies xs ! i' = \text{None}$
and $ii < jj$
and $i < \text{length } xs$ **and** $\text{mk-eqcl}' xs ii jj l T ! i = \text{Some } m$
and $j < \text{length } xs$
shows $(\text{mk-eqcl}' xs ii jj l T ! j = \text{Some } m) = (\neg \text{tr-lookup } T (i + jj) (j + jj))$
using *assms* **proof** (*induct xs arbitrary: jj i j*)
case *Nil*
from $\text{Nil}(7)$ **have** *False* **by** *simp*
thus $?case$ **by** *simp*
next
case $(\text{Cons } y xs jj i j)$
show $?case$ **proof** (*cases i*)
case 0
show $?thesis$ **proof** (*cases j*)
case 0
with $\langle i=0 \rangle$ $\text{Cons}(9)$ **show** $?thesis$ **by** (*simp add: tr-lookup-def*)
next
case $(\text{Suc } j')$
from 0 $\text{Cons}(5,9)$ **have** 1: $y = \text{Some } m \wedge m < l \vee (y = \text{None} \wedge \neg \text{tr-lookup } T jj ii \wedge m = l)$ **by** (*cases y, cases tr-lookup T jj ii, auto*)
thus $?thesis$ **proof** (*elim disjE*)
assume $H: y = \text{Some } m \wedge m < l$
from Suc **have** $(\text{mk-eqcl}' (y \# xs) ii jj l T ! j = \text{Some } m) = (\text{mk-eqcl}' xs ii (\text{Suc } jj) l T ! j' = \text{Some } m)$ **by** *simp*
also from H **have** $\dots = (xs ! j' = \text{Some } m)$ **proof** (*induct xs arbitrary: jj j'*)

case (*Cons a xs jj j'*) **thus** ?*case* **by** (*cases j'*) *simp+*
qed *simp*
also from *Suc* **have** ... = ((*y # xs*) ! *j = Some m*) **by** *simp*
also from *Cons(2)[of i j m]* *Cons(8,10)* *Suc 0 H* **have** ... = (\neg *tr-lookup T* (*i + jj*) (*j + jj*)) **by** *simp*
finally show ?*thesis* **by** *simp*
next
assume *H*: *y = None* \wedge \neg *tr-lookup T jj ii* \wedge *m = l*
with *Suc* **have** (*mk-eqcl'* (*y # xs*) *ii jj l T* ! *j = Some m*) = (*mk-eqcl'* *xs ii* (*Suc jj*) *l T* ! *j' = Some l*) **by** *simp*
also have ... = (\neg *tr-lookup T* (*j' + Suc jj*) *ii*) **proof** (*rule mk-eqcl'-nth'*)
from *Cons(5)* **show** $\bigwedge x k.$ $\llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < l$ **by** *simp*
show $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + \text{Suc } jj) \text{ ii} \rrbracket \implies xs ! i' =$
None **proof** –
fix *i'* **assume** *i' < length xs* \neg *tr-lookup T (i' + Suc jj) ii*
with *Cons(6)[of Suc i']* **show** *xs ! i' = None* **by** *simp*
qed
from *Cons(7)* **show** *ii < Suc jj* **by** *simp*
from *Cons(10)* *Suc* **show** *j' < length xs* **by** *simp*
qed
also from *Suc H 0* **have** ... = (\neg *tr-lookup T* (*j + jj*) *ii* \wedge \neg *tr-lookup T* (*i + jj*) *ii*) **by** (*simp add: add commute*)
also have ... = (\neg *tr-lookup T* (*i + jj*) (*j + jj*) \wedge \neg *tr-lookup T* (*i + jj*) *ii*) **proof**
assume *H'*: \neg *tr-lookup T* (*j + jj*) *ii* \wedge \neg *tr-lookup T* (*i + jj*) *ii*
hence \neg *tr-lookup T* *ii* (*j + jj*) **by** (*auto simp: tr-lookup-def*)
with *H'* *Cons(3)[of i + jj ii j + jj]* *Cons(4,7,8,10)* **show** \neg *tr-lookup T* (*i + jj*) (*j + jj*) \wedge \neg *tr-lookup T* (*i + jj*) *ii* **by** *simp*
next
assume *H'*: \neg *tr-lookup T* (*i + jj*) (*j + jj*) \wedge \neg *tr-lookup T* (*i + jj*) *ii*
hence \neg *tr-lookup T* (*j + jj*) (*i + jj*) **by** (*auto simp: tr-lookup-def*)
with *H'* *Cons(3)[of j + jj i + jj ii]* *Cons(4,7,8,10)* **show** \neg *tr-lookup T* (*j + jj*) *ii* \wedge \neg *tr-lookup T* (*i + jj*) *ii* **by** *simp*
qed
also from *0 H* **have** ... = (\neg *tr-lookup T* (*i + jj*) (*j + jj*)) **by** *simp*
finally show ?*thesis* **by** *simp*
qed
qed
next
case (*Suc i'*)
show ?*thesis* **proof** (*cases j*)
case *0*
have *m* \leq *l* **proof** (*rule mk-eqcl'-bound*)
from *Cons(5)* **show** $\bigwedge x k.$ $\llbracket x \in \text{set } (y \# xs); x = \text{Some } k \rrbracket \implies k < l$ **by**
simp
from *Cons(8)* **have** *i < length (mk-eqcl' (y # xs) ii jj l T)* **by** (*simp add: mk-eqcl'-len*)
with *Cons(9)* **have** $\exists i < \text{length } (mk-eqcl' (y \# xs) ii jj l T).$ *mk-eqcl' (y # xs) ii jj l T ! i = Some m* **by** *blast*

thus $\text{Some } m \in \text{set } (mk\text{-eqcl}' (y \# xs) \text{ ii } jj \text{ l } T)$ **by** (*simp only: in-set-conv-nth*)
show $\text{Some } m = \text{Some } m$ **by** *simp*
qed
hence $m < l \vee m = l$ **by** *auto*
thus *?thesis* **proof** (*elim disjE*)
assume $H: m < l$
with *Cons(9)* **have** $I: (y \# xs) ! i = \text{Some } m$ **proof** (*induct (y \# xs)*
arbitrary: jj i)
case (*Cons a l jj i*) **thus** *?case* **by** (*cases i*) (*auto, cases tr-lookup T jj ii*
 $\vee a \neq \text{None}, \text{simp+}$)
qed *simp*
from $0 H$ **have** $(mk\text{-eqcl}' (y \# xs) \text{ ii } jj \text{ l } T ! j = \text{Some } m) = ((y \# xs) ! j =$
 $\text{Some } m)$ **by** (*cases tr-lookup T jj ii* $\vee y \neq \text{None}$) *simp+*
also from *Cons(8,10)* I **have** $\dots = (\neg \text{tr-lookup } T (i + jj) (j + jj))$ **by**
(*rule Cons(2)*)
finally show *?thesis* **by** *simp*
next
assume $H: m = l$
from *Cons(5,6,7,8)* **have** $(mk\text{-eqcl}' (y \# xs) \text{ ii } jj \text{ l } T ! i = \text{Some } l) = (\neg$
 $\text{tr-lookup } T (i + jj) \text{ ii})$ **by** (*rule mk-eqcl'-nth'*)
with H *Cons(9)* **have** $I: \neg \text{tr-lookup } T (i + jj) \text{ ii}$ **by** *simp*

with $0 H$ *Cons(5)* **have** $(mk\text{-eqcl}' (y \# xs) \text{ ii } jj \text{ l } T ! j = \text{Some } m) = (\neg$
 $\text{tr-lookup } T (j + jj) \text{ ii} \wedge \neg \text{tr-lookup } T (i + jj) \text{ ii} \wedge y = \text{None})$ **by** *auto*
also from *Cons(6)[of 0]* 0 **have** $\dots = (\neg \text{tr-lookup } T (j + jj) \text{ ii} \wedge \neg \text{tr-lookup}$
 $T (i + jj) \text{ ii})$ **by** *auto*
also have $\dots = (\neg \text{tr-lookup } T (i + jj) (j + jj) \wedge \neg \text{tr-lookup } T (i + jj)$
 $\text{ii})$ **proof**
assume $H': \neg \text{tr-lookup } T (j + jj) \text{ ii} \wedge \neg \text{tr-lookup } T (i + jj) \text{ ii}$
hence $\neg \text{tr-lookup } T \text{ ii } (j + jj)$ **by** (*auto simp: tr-lookup-def*)
with H' *Cons(3)[of i + jj ii j + jj]* *Cons(4,7,8,10)* **show** $\neg \text{tr-lookup } T (i$
 $+ jj) (j + jj) \wedge \neg \text{tr-lookup } T (i + jj) \text{ ii}$ **by** *simp*
next
assume $H': \neg \text{tr-lookup } T (i + jj) (j + jj) \wedge \neg \text{tr-lookup } T (i + jj) \text{ ii}$
hence $\neg \text{tr-lookup } T (j + jj) (i + jj)$ **by** (*auto simp: tr-lookup-def*)
with H' *Cons(3)[of j + jj i + jj ii]* *Cons(4,7,8,10)* **show** $\neg \text{tr-lookup } T (j$
 $+ jj) \text{ ii} \wedge \neg \text{tr-lookup } T (i + jj) \text{ ii}$ **by** *simp*
qed
also from I **have** $\dots = (\neg \text{tr-lookup } T (i + jj) (j + jj))$ **by** *simp*
finally show *?thesis* **by** *simp*
qed
next
case (*Suc j'*)
hence $(mk\text{-eqcl}' (y \# xs) \text{ ii } jj \text{ l } T ! j = \text{Some } m) = (mk\text{-eqcl}' xs \text{ ii } (\text{Suc } jj) \text{ l}$
 $T ! j' = \text{Some } m)$ **by** *simp*
also have $\dots = (\neg \text{tr-lookup } T (i' + \text{Suc } jj) (j' + \text{Suc } jj))$ **proof** (*rule*
Cons(1))
show $\bigwedge i' j' k. \llbracket i' < \text{length } xs; j' < \text{length } xs; xs ! i' = \text{Some } k \rrbracket \implies (xs ! j'$
 $= \text{Some } k) = (\neg \text{tr-lookup } T (i' + \text{Suc } jj) (j' + \text{Suc } jj))$ **proof** –

fix $i' j' k$ **assume** $i' < \text{length } xs \ j' < \text{length } xs \ xs ! i' = \text{Some } k$
with $\text{Cons}(2)[\text{of } \text{Suc } i' \ \text{Suc } j' \ k]$ **show** $(xs ! j' = \text{Some } k) = (\neg \text{tr-lookup } T (i' + \text{Suc } jj) (j' + \text{Suc } jj))$ **by** *simp*
qed
from $\text{Cons}(3)$ **show** $\bigwedge a \ b \ c. \llbracket a \leq \text{length } T; b \leq \text{length } T; c \leq \text{length } T; \neg \text{tr-lookup } T \ a \ b; \neg \text{tr-lookup } T \ b \ c \rrbracket \implies \neg \text{tr-lookup } T \ a \ c$ **by** *blast*
from $\text{Cons}(4)$ **show** $\text{length } xs + \text{Suc } jj = \text{length } T + 1$ **by** *simp*
from $\text{Cons}(5)$ **show** $\bigwedge x \ k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < l$ **by** *simp*
show $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + \text{Suc } jj) \ ii \rrbracket \implies xs ! i' = \text{None}$
proof –
fix i' **assume** $i' < \text{length } xs \ \neg \text{tr-lookup } T (i' + \text{Suc } jj) \ ii$
with $\text{Cons}(6)[\text{of } \text{Suc } i']$ **show** $xs ! i' = \text{None}$ **by** *simp*
qed
from $\text{Cons}(7)$ **show** $ii < \text{Suc } jj$ **by** *simp*
from $\text{Cons}(8) \ \langle i = \text{Suc } i' \rangle$ **show** $i' < \text{length } xs$ **by** *simp*
from $\text{Cons}(9) \ \langle i = \text{Suc } i' \rangle$ **show** $mk\text{-eqcl}' \ xs \ ii \ (\text{Suc } jj) \ l \ T ! i' = \text{Some } m$
by *simp*
from $\text{Cons}(10) \ \text{Suc}$ **show** $j' < \text{length } xs$ **by** *simp*
qed
also from $\text{Suc} \ \langle i = \text{Suc } i' \rangle$ **have** $\dots = (\neg \text{tr-lookup } T (i + jj) (j + jj))$ **by** *simp*
finally show *?thesis* **by** *simp*
qed
qed
qed

lemma *mk-eqcl'-Some*:
assumes $i < \text{length } xs$ **and** $xs ! i \neq \text{None}$
shows $mk\text{-eqcl}' \ xs \ ii \ j \ l \ T ! i = xs ! i$
using *assms* **proof** (*induct xs arbitrary: j i*)
case $(\text{Cons } y \ xs \ j \ i)$
thus *?case* **by** (*cases i*) *auto*
qed *simp*

lemma *mk-eqcl'-Some2*:
assumes $i < \text{length } xs$
and $k < l$
shows $(mk\text{-eqcl}' \ xs \ ii \ j \ l \ T ! i = \text{Some } k) = (xs ! i = \text{Some } k)$
using *assms* **proof** (*induct xs arbitrary: j i*)
case $(\text{Cons } y \ xs \ j \ i)$
thus *?case* **by** (*cases i*) *auto*
qed *simp*

lemma *mk-eqcl-fst-Some*:
assumes $i < \text{length } xs$ **and** $k < \text{length } zs$
shows $(fst (mk\text{-eqcl } xs \ zs \ ii \ T) ! i = k) = (xs ! i = \text{Some } k)$
using *assms* **proof** (*induct xs zs ii T arbitrary: i rule: mk-eqcl.induct*)
case $(2 \ xs \ zs \ ii \ T \ i)$
thus *?case* **by** (*cases i*) (*simp add: split-beta mk-eqcl'-len mk-eqcl'-Some2*)+

next
case (3 l xs zs ii T i)
thus ?case **by** (cases i) (simp add: split-beta)+
qed simp

lemma mk-egcl-len-snd:
length zs ≤ length (snd (mk-egcl xs zs i T))
by (induct xs zs i T rule: mk-egcl.induct) (simp add: split-beta)+

lemma mk-egcl-len-fst:
length (fst (mk-egcl xs zs i T)) = length xs
by (induct xs zs i T rule: mk-egcl.induct) (simp add: split-beta mk-egcl'-len)+

lemma mk-egcl-set-snd:
assumes i ∉ set zs
and j > i
shows i ∉ set (snd (mk-egcl xs zs j T))
using *assms* **by** (induct xs zs j T rule: mk-egcl.induct) (auto simp: split-beta)

lemma mk-egcl-snd-mon:
assumes ∧j1 j2. [j1 < j2; j2 < length zs] ⇒ zs ! j1 < zs ! j2
and ∧x. x ∈ set zs ⇒ x < i
and j1 < j2 **and** j2 < length (snd (mk-egcl xs zs i T))
shows snd (mk-egcl xs zs i T) ! j1 < snd (mk-egcl xs zs i T) ! j2
using *assms* **proof** (induct xs zs i T rule: mk-egcl.induct)
case (2 xs zs i T)
have ∧j1 j2. [j1 < j2; j2 < length (zs @ [i])] ⇒ (zs @ [i]) ! j1 < (zs @ [i]) ! j2
j2 proof –
fix j1 j2 **assume** H: j1 < j2 j2 < length (zs @ [i])
hence j2 < length zs ∨ j2 = length zs **by** auto
from this H 2 **show** (zs @ [i]) ! j1 < (zs @ [i]) ! j2 **by** (elim disjE) (simp add: nth-append)+
qed moreover
have ∧x. x ∈ set (zs @ [i]) ⇒ x < Suc i **proof** –
fix x **assume** x ∈ set (zs @ [i])
hence x ∈ set zs ∨ x = i **by** auto
with 2(3)[of x] **show** x < Suc i **by** auto
qed moreover
note 2(4) **moreover**
from 2(5) **have** j2 < length (snd (mk-egcl (mk-egcl' xs i (Suc i) (length zs) T) (zs @ [i]) (Suc i) T)) **by** (simp add: split-beta)
ultimately **have** snd (mk-egcl (mk-egcl' xs i (Suc i) (length zs) T) (zs @ [i]) (Suc i) T) ! j1 < snd (mk-egcl (mk-egcl' xs i (Suc i) (length zs) T) (zs @ [i]) (Suc i) T) ! j2 **by** (rule 2(1))
thus ?case **by** (simp add: split-beta)

next
case (3 l xs zs i T)
note 3(2) **moreover**
have ∧x. x ∈ set zs ⇒ x < Suc i **proof** –

```

    fix x assume x ∈ set zs
    with 3(3)[of x] show x < Suc i by simp
  qed moreover
  note 3(4) moreover
  from 3(5) have j2 < length (snd (mk-eqcl xs zs (Suc i) T)) by (simp add:
split-beta)
  ultimately have snd (mk-eqcl xs zs (Suc i) T) ! j1 < snd (mk-eqcl xs zs (Suc
i) T) ! j2 by (rule 3(1))
  thus ?case by (simp add: split-beta)
qed simp

lemma mk-eqcl-snd-nth:
  assumes i < length zs
  shows snd (mk-eqcl xs zs j T) ! i = zs ! i
using assms by (induct xs zs j T rule: mk-eqcl.induct) (simp add: split-beta nth-append)+

lemma mk-eqcl-bound:
  assumes  $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < \text{length } zs$ 
  and  $x \in \text{set } (\text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } ii \text{ } T))$ 
  shows  $x < \text{length } (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } ii \text{ } T))$ 
using assms proof (induct xs zs ii T rule: mk-eqcl.induct)
  case (2 xs zs i T)
  hence  $x = \text{length } zs \vee x \in \text{set } (\text{fst } (\text{mk-eqcl } (\text{mk-eqcl}' xs \text{ } i \text{ } (\text{Suc } i) \text{ } (\text{length } zs) \text{ } T) (\text{zs } @ [i]) (\text{Suc } i) \text{ } T))$  by (auto simp: split-beta)
  thus ?case proof (elim disjE)
    assume  $x = \text{length } zs$ 
    hence  $x < \text{length } (\text{zs } @ [i])$  by simp
    also have  $\dots \leq \text{length } (\text{snd } (\text{mk-eqcl } (\text{mk-eqcl}' xs \text{ } i \text{ } (\text{Suc } i) \text{ } (\text{length } zs) \text{ } T) (\text{zs } @ [i]) (\text{Suc } i) \text{ } T))$  by (simp only: mk-eqcl-len-snd)
    finally show ?thesis by (simp add: split-beta)
  next
    assume  $H: x \in \text{set } (\text{fst } (\text{mk-eqcl } (\text{mk-eqcl}' xs \text{ } i \text{ } (\text{Suc } i) \text{ } (\text{length } zs) \text{ } T) (\text{zs } @ [i]) (\text{Suc } i) \text{ } T))$ 
    have  $\bigwedge x k. \llbracket x \in \text{set } (\text{mk-eqcl}' xs \text{ } i \text{ } (\text{Suc } i) \text{ } (\text{length } zs) \text{ } T); x = \text{Some } k \rrbracket \implies k < \text{length } (\text{zs } @ [i])$  proof -
      fix x k assume  $H': x \in \text{set } (\text{mk-eqcl}' xs \text{ } i \text{ } (\text{Suc } i) \text{ } (\text{length } zs) \text{ } T) \ x = \text{Some } k$ 
      { fix x' k' assume  $x' \in \text{set } xs \ x' = \text{Some } k'$ 
        with 2 have  $k' < \text{length } zs$  by simp
      } from this H' have  $k \leq \text{length } zs$  by (rule mk-eqcl'-bound)
      thus  $k < \text{length } (\text{zs } @ [i])$  by simp
    qed
  with H 2 show ?thesis by (simp add: split-beta)
qed
next
case (3 l xs zs i T)
hence  $x = l \vee x \in \text{set } (\text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T))$  by (auto simp: split-beta)
thus ?case proof (elim disjE)
  assume  $x = l$ 
  with 3 have  $x < \text{length } zs$  by simp

```

also from 3 have $\dots \leq \text{length} (\text{snd} (\text{mk-eqcl} (\text{Some } l \# xs) zs \ i \ T))$ **by** (*simp only: mk-eqcl-len-snd*)
finally show *?thesis* **by** *simp*
next
assume $x \in \text{set} (\text{fst} (\text{mk-eqcl} \ xs \ zs \ (\text{Suc } i) \ T))$
with 3 have $x < \text{length} (\text{snd} (\text{mk-eqcl} \ xs \ zs \ (\text{Suc } i) \ T))$ **by** *simp*
thus *?thesis* **by** (*simp add: split-beta*)
qed
qed *simp*

lemma *mk-eqcl-fst-snd*:

assumes $\bigwedge i. i < \text{length} \ zs \implies zs \ ! \ i < \text{length} \ xs + ii \wedge (zs \ ! \ i \geq ii \longrightarrow xs \ ! \ (zs \ ! \ i - ii) = \text{Some } i)$
and $\bigwedge j1 \ j2. [j1 < j2; j2 < \text{length} \ zs] \implies zs \ ! \ j1 < zs \ ! \ j2$
and $\bigwedge z. z \in \text{set} \ zs \implies z < ii$
and $i < \text{length} (\text{snd} (\text{mk-eqcl} \ xs \ zs \ ii \ T))$
and $\text{length} \ xs + ii \leq \text{length} \ T + 1$
shows $\text{snd} (\text{mk-eqcl} \ xs \ zs \ ii \ T) \ ! \ i < \text{length} (\text{fst} (\text{mk-eqcl} \ xs \ zs \ ii \ T)) + ii \wedge (\text{snd} (\text{mk-eqcl} \ xs \ zs \ ii \ T) \ ! \ i \geq ii \longrightarrow \text{fst} (\text{mk-eqcl} \ xs \ zs \ ii \ T) \ ! \ (\text{snd} (\text{mk-eqcl} \ xs \ zs \ ii \ T) \ ! \ i - ii) = i)$
using *assms* **proof** (*induct xs zs ii T arbitrary: i rule: mk-eqcl.induct*)
case (1 *zs ii T i*)
from 1(1)[*of i*] 1(4,5) **show** *?case* **by** *simp*
next
case (2 *xs zs i T j*)
have $\bigwedge i'. i' < \text{length} (zs \ @ \ [i]) \implies (zs \ @ \ [i]) \ ! \ i' < \text{length} (\text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T) + \text{Suc } i \wedge$
 $(\text{Suc } i \leq (zs \ @ \ [i]) \ ! \ i' \longrightarrow \text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T \ ! \ ((zs \ @ \ [i]) \ ! \ i' - \text{Suc } i) = \text{Some } i')$ **proof** -
fix *i'* **assume** $i' < \text{length} (zs \ @ \ [i])$
hence $i' < \text{length} \ zs \vee i' = \text{length} \ zs$ **by** *auto*
thus $(zs \ @ \ [i]) \ ! \ i' < \text{length} (\text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T) + \text{Suc } i \wedge$
 $(\text{Suc } i \leq (zs \ @ \ [i]) \ ! \ i' \longrightarrow \text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T \ ! \ ((zs \ @ \ [i]) \ ! \ i' - \text{Suc } i) = \text{Some } i')$
proof (*elim disjE*)
assume $H: i' < \text{length} \ zs$
with 2(2) **have** $I: zs \ ! \ i' < \text{length} (\text{None} \ # \ xs) + i \wedge (i \leq zs \ ! \ i' \longrightarrow (\text{None} \ # \ xs) \ ! \ (zs \ ! \ i' - i) = \text{Some } i')$ **by** *simp*
with H **have** $G1: (zs \ @ \ [i]) \ ! \ i' < \text{length} (\text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T) + \text{Suc } i$ **by** (*auto simp: mk-eqcl'-len nth-append*)
{ **assume** $H': \text{Suc } i \leq (zs \ @ \ [i]) \ ! \ i'$
then obtain k **where** $K: (zs \ @ \ [i]) \ ! \ i' - i = \text{Suc } k$ **by** (*cases (zs @ [i]) ! i' - i simp+*)
hence $K': k = (zs \ @ \ [i]) \ ! \ i' - \text{Suc } i$ **by** *simp*
from $K \ H' \ H \ I$ **have** $xs \ ! \ k = \text{Some } i'$ **by** (*simp add: nth-append*)
with $K \ I \ H$ **have** $\text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T \ ! \ k = \text{Some } i'$ **by** (*auto simp add: mk-eqcl'-Some nth-append*)
with K' **have** $\text{mk-eqcl}' \ xs \ i \ (\text{Suc } i) (\text{length} \ zs) \ T \ ! \ ((zs \ @ \ [i]) \ ! \ i' - \text{Suc } i) = \text{Some } i'$ **by** *simp*

} with $G1$ show $?thesis$ by *simp*
qed *simp*
qed
moreover have $\bigwedge j1\ j2. \llbracket j1 < j2; j2 < \text{length } (zs @ [i]) \rrbracket \implies (zs @ [i]) ! j1 < (zs @ [i]) ! j2$ proof –
fix $j1\ j2$ assume $H: j1 < j2\ j2 < \text{length } (zs @ [i])$
hence $j2 < \text{length } zs \vee j2 = \text{length } zs$ by *auto*
from *this* $H\ 2(3)[\text{of } j1\ j2]\ 2(4)[\text{of } zs ! j1]$ show $(zs @ [i]) ! j1 < (zs @ [i]) ! j2$
by (*elim disjE*) (*simp add: nth-append*)
qed
moreover have $\bigwedge z. z \in \text{set } (zs @ [i]) \implies z < \text{Suc } i$ proof –
fix z assume $z \in \text{set } (zs @ [i])$
hence $z \in \text{set } zs \vee z = i$ by *auto*
with $2(4)[\text{of } z]$ show $z < \text{Suc } i$ by *auto*
qed
moreover from 2 have $j < \text{length } (\text{snd } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T))$ by (*simp add: split-beta*)
moreover from 2 have $\text{length } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T) + \text{Suc } i \leq \text{length } T + 1$ by (*simp add: mk-eqcl'-len*)
ultimately have $IV: \text{snd } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T) ! j < \text{length } (\text{fst } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T)) + \text{Suc } i \wedge$
 $(\text{Suc } i \leq \text{snd } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T) ! j \longrightarrow$
 $\text{fst } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T) ! (\text{snd } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T) ! j - \text{Suc } i) = j$ by (*rule 2(1)*)
hence $G1: \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j < \text{length } (\text{fst } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T)) + i$ by (*auto simp: split-beta*)
{ assume $i \leq \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j$
hence $i = \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j \vee \text{Suc } i \leq \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j$ by *auto*
hence $\text{fst } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! (\text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j - i) = j$ proof (*elim disjE*)
assume $H: i = \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j$
define k where $k = \text{length } zs$
hence $K: \text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! k = i$ by (*simp add: mk-eqcl-snd-nth split-beta*)
{ assume $j \neq k$
hence $j < k \vee j > k$ by *auto*
hence $\text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j \neq i$ proof (*elim disjE*)
assume $H': j < k$
from $k\text{-def}$ have $k < \text{length } (zs @ [i])$ by *simp*
also have $\dots \leq \text{length } (\text{snd } (\text{mk-eqcl } (\text{mk-eqcl}'\ xs\ i\ (\text{Suc } i)\ (\text{length } zs)\ T)\ (zs @ [i])\ (\text{Suc } i)\ T))$ by (*simp only: mk-eqcl-len-snd*)
also have $\dots = \text{length } (\text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T))$ by (*simp add: split-beta*)
finally have $K': k < \text{length } (\text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T))$ by *simp*
from $2(3,4)\ H'$ *this* have $\text{snd } (\text{mk-eqcl } (\text{None } \# xs)\ zs\ i\ T) ! j < \text{snd}$

```

(mk-eqcl (None # xs) zs i T) ! k by (rule mk-eqcl-snd-mon)
  with K show ?thesis by simp
next
  assume H': j > k
  from 2(3,4) H' 2(5) have snd (mk-eqcl (None # xs) zs i T) ! k < snd
(mk-eqcl (None # xs) zs i T) ! j by (rule mk-eqcl-snd-mon)
  with K show ?thesis by simp
  qed
}
with H k-def have j = length zs by auto
with H show ?thesis by (simp add: split-beta)
next
  assume H: Suc i ≤ snd (mk-eqcl (None # xs) zs i T) ! j
  then obtain k where K: snd (mk-eqcl (None # xs) zs i T) ! j - i = Suc k
by (cases snd (mk-eqcl (None # xs) zs i T) ! j - i) simp+
  hence K': k = snd (mk-eqcl (None # xs) zs i T) ! j - Suc i by simp
  from H IV have fst (mk-eqcl (mk-eqcl' xs i (Suc i) (length zs) T) (zs @ [i])
(Suc i) T) ! (snd (mk-eqcl (mk-eqcl' xs i (Suc i) (length zs) T) (zs @ [i]) (Suc i)
T) ! j - Suc i) = j
  by (auto simp: split-beta)
  with K' have fst (mk-eqcl (None # xs) zs i T) ! Suc k = j by (simp add:
split-beta)
  with K show ?thesis by simp
  qed
} with G1 show ?case by simp
next
  case (3 l xs zs i T j)
  have 1: snd (mk-eqcl (Some l # xs) zs i T) = snd (mk-eqcl xs zs (Suc i) T) by
(simp add: split-beta)
  have 2: length (fst (mk-eqcl (Some l # xs) zs i T)) = length (Some l # xs) by
(simp add: split-beta mk-eqcl-len-fst)
  have  $\bigwedge j. j < \text{length } zs \implies zs ! j < \text{length } xs + \text{Suc } i \wedge (\text{Suc } i \leq zs ! j \longrightarrow xs !
(zs ! j - \text{Suc } i) = \text{Some } j)$  proof -
  fix j assume H: j < length zs
  with 3(2)[of j] have I: zs ! j < length (Some l # xs) + i  $\wedge$  (i ≤ zs ! j  $\longrightarrow$ 
(Some l # xs) ! (zs ! j - i) = Some j) by simp
  hence G1: zs ! j < length xs + Suc i and G2: i ≤ zs ! j  $\longrightarrow$  (Some l # xs) !
(zs ! j - i) = Some j by simp+
  { assume H2: Suc i ≤ zs ! j
    then obtain k where K: zs ! j - i = Suc k by (cases zs ! j - i) simp+
    with H2 G2 have xs ! k = Some j by simp
    moreover from K have k = zs ! j - Suc i by simp
    ultimately have xs ! (zs ! j - Suc i) = Some j by simp
  }
  with G1 show zs ! j < length xs + Suc i  $\wedge$  (Suc i ≤ zs ! j  $\longrightarrow$  xs ! (zs ! j -
Suc i) = Some j) by simp
  qed
moreover note 3(3)
moreover have  $\bigwedge z. z \in \text{set } zs \implies z < \text{Suc } i$  proof -

```

fix z **assume** $z \in \text{set } zs$
with 3(4)[of z] **show** $z < \text{Suc } i$ **by** *simp*
qed
moreover from 3(5) 1 **have** $j < \text{length } (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T))$ **by** *simp*
moreover from 3 **have** $\text{length } xs + \text{Suc } i \leq \text{length } T + 1$ **by** *simp*
ultimately have $IV: \text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! j < \text{length } (\text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T)) + \text{Suc } i \wedge$
 $(\text{Suc } i \leq \text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! j \longrightarrow \text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) !$
 $(\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! j - \text{Suc } i) = j)$ **by** (rule 3(1))
with 1 **have** $G1: \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j < \text{length } (\text{fst } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T)) + i$ **by** (*simp add: split-beta mk-eqcl-len-fst*)
{ assume $i \leq \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j$
hence $i = \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j \vee i < \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j$ **by** *auto*
hence $\text{fst } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! (\text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j - i) = j$ **proof** (*elim disjE*)
assume $H: i = \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j$
with 3 1 **have** $\exists j < \text{length } (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T)). \text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! j = i$ **by** *auto*
hence $T1: i \in \text{set } (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T))$ **by** (*simp only: in-set-conv-nth*)
from 3(4) **have** $i \notin \text{set } zs$ **by** *auto*
hence $i \notin \text{set } (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T))$ **by** (*simp add: mk-eqcl-set-snd*)
with $T1$ **show** *?thesis* **by** *simp*
next
assume $H: i < \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j$
from H **obtain** k **where** $K: \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j - i = \text{Suc } k$ **by** (*cases snd (mk-eqcl (Some l # xs) zs i T) ! j - i simp+*)
hence $K': \text{snd } (\text{mk-eqcl } (\text{Some } l \# xs) \text{ } zs \text{ } i \text{ } T) ! j - \text{Suc } i = k$ **by** *simp*
from 1 H IV **have** $\text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! (\text{snd } (\text{mk-eqcl } xs \text{ } zs \text{ } (\text{Suc } i) \text{ } T) ! j - \text{Suc } i) = j$ **by** *simp*
with K K' **show** *?thesis* **by** (*simp add: split-beta*)
qed
} with $G1$ **show** *?case* **by** *simp*
qed

lemma *mk-eqcl-fst-nth*:

assumes $\bigwedge i j k. \llbracket i < \text{length } xs; j < \text{length } xs; xs ! i = \text{Some } k \rrbracket \Longrightarrow (xs ! j = \text{Some } k) = (\neg \text{tr-lookup } T (i + ii) (j + ii))$
and $\bigwedge a b c. \llbracket a \leq \text{length } T; b \leq \text{length } T; c \leq \text{length } T; \neg \text{tr-lookup } T a b; \neg \text{tr-lookup } T b c \rrbracket \Longrightarrow \neg \text{tr-lookup } T a c$
and $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \Longrightarrow k < \text{length } xs$
and $\text{length } xs + ii = \text{length } T + 1$
and $i < \text{length } xs$ **and** $j < \text{length } xs$
shows $(\text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } ii \text{ } T) ! i = \text{fst } (\text{mk-eqcl } xs \text{ } zs \text{ } ii \text{ } T) ! j) = (\neg \text{tr-lookup } T (i + ii) (j + ii))$
using *assms* **proof** (*induct xs zs ii T arbitrary: i j rule: mk-eqcl.induct*)
case (1 $zs \text{ } ii \text{ } T$) **thus** *?case* **by** *simp*
next
case (2 $xs \text{ } zs \text{ } ii \text{ } T$)

{ fix $i j$ assume $H: i < j j < \text{length} (None \# xs)$
then obtain j' where $J: j = \text{Suc } j'$ by (cases j) simp+
have $(fst (mk\text{-}eqcl (None \# xs) zs \ ii \ T) ! i = fst (mk\text{-}eqcl (None \# xs) zs \ ii \ T) ! j) = (\neg \text{tr}\text{-}lookup \ T \ (i + ii) \ (j + ii))$ **proof (cases i)**
case 0
with J have $(fst (mk\text{-}eqcl (None \# xs) zs \ ii \ T) ! i = fst (mk\text{-}eqcl (None \# xs) zs \ ii \ T) ! j) = (fst (mk\text{-}eqcl (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T) (zs \ @ \ [ii]) (\text{Suc } ii) \ T) ! j' = \text{length } zs)$
by (auto simp add: split-beta)
also from $H J$ have $\dots = (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T ! j' = \text{Some} (\text{length } zs))$ **by (simp add: mk-eqcl-fst-Some mk-eqcl'-len)
also have $\dots = (\neg \text{tr}\text{-}lookup \ T \ (j' + \text{Suc } ii) \ ii)$ **proof –**
have $\bigwedge x k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < \text{length } zs$ **proof –**
fix $x k$ assume $x \in \text{set } xs \ x = \text{Some } k$
with 2(4)[of $x k$] show $k < \text{length } zs$ **by simp**
qed moreover
have $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr}\text{-}lookup \ T \ (i' + \text{Suc } ii) \ ii \rrbracket \implies xs ! i' = \text{None}$
proof –
fix i' assume $H: i' < \text{length } xs \ \neg \text{tr}\text{-}lookup \ T \ (i' + \text{Suc } ii) \ ii$
{ assume $H': xs ! i' \neq \text{None}$
then obtain k where $xs ! i' = \text{Some } k$ **by (cases $xs ! i'$) simp+**
with 2(2)[of $\text{Suc } i' \ 0 \ k$] H have False **by simp**
} thus $xs ! i' = \text{None}$ **by (cases $xs ! i'$) simp+**
qed moreover
from $H J$ have $ii < \text{Suc } ii \ j' < \text{length } xs$ **by simp+**
ultimately show ?thesis **by (rule mk-eqcl'-nth')**
qed
also from $J \ 0$ have $\dots = (\neg \text{tr}\text{-}lookup \ T \ (i + ii) \ (j + ii))$ **by (auto simp: tr-lookup-def)
finally show ?thesis **by simp**
next
case (Suc i')
have $\bigwedge i j k. \llbracket i < \text{length} (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T); j < \text{length} (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T); mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T ! i = \text{Some } k \rrbracket$
 $\implies (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T ! j = \text{Some } k) = (\neg \text{tr}\text{-}lookup \ T \ (i + \text{Suc } ii) \ (j + \text{Suc } ii))$ **proof –**
fix $i j k$ assume $H: i < \text{length} (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T) \ j < \text{length} (mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T) \ mk\text{-}eqcl' xs \ ii \ (\text{Suc } ii) \ (\text{length } zs) \ T ! i = \text{Some } k$
{ fix $i' j' k$ assume $i' < \text{length } xs \ j' < \text{length } xs \ xs ! i' = \text{Some } k$
with 2(2)[of $\text{Suc } i' \ \text{Suc } j' \ k$] have $(xs ! j' = \text{Some } k) = (\neg \text{tr}\text{-}lookup \ T \ (i' + \text{Suc } ii) \ (j' + \text{Suc } ii))$ **by simp**
} moreover
note 2(3) moreover
from 2(5) have $\text{length } xs + \text{Suc } ii = \text{length } T + 1$ **by simp moreover**
{ fix $x k$ assume $x \in \text{set } xs \ x = \text{Some } k$
with 2(4)[of $x k$] have $k < \text{length } zs$ **by simp**
} moreover****

have $\bigwedge i'. \llbracket i' < \text{length } xs; \neg \text{tr-lookup } T (i' + \text{Suc } ii) \text{ } ii \rrbracket \implies xs ! i' = \text{None}$
proof –
 fix i' assume $H': i' < \text{length } xs \neg \text{tr-lookup } T (i' + \text{Suc } ii) \text{ } ii$
 { assume $xs ! i' \neq \text{None}$
 then obtain k where $K: xs ! i' = \text{Some } k$ by (cases $xs ! i'$) simp+
 with $H' 2(2)[\text{of } \text{Suc } i' 0 k]$ have False by simp
 } thus $xs ! i' = \text{None}$ by (cases $xs ! i' = \text{None}$) simp+
qed moreover
 have $ii < \text{Suc } ii$ by simp moreover
 from H have $i < \text{length } xs$ by (simp add: mk-eqcl'-len) moreover
 note $H(3)$ moreover
 from H have $j < \text{length } xs$ by (simp add: mk-eqcl'-len)
 ultimately show $(\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T ! j = \text{Some } k) = (\neg$
 $\text{tr-lookup } T (i + \text{Suc } ii) (j + \text{Suc } ii))$ by (rule mk-eqcl'-nth)
qed moreover
 note $2(3)$ moreover
 have $\bigwedge x k. \llbracket x \in \text{set } (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T); x = \text{Some } k \rrbracket \implies$
 $k < \text{length } (zs @ [ii])$ **proof** –
 fix $x k$ assume $H: x \in \text{set } (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T) x = \text{Some}$
 k
 { fix $x k$ assume $x \in \text{set } xs x = \text{Some } k$
 with $2(4)[\text{of } x k]$ have $k < \text{length } zs$ by simp
 } from $\text{this } H$ have $k \leq \text{length } zs$ by (rule mk-eqcl'-bound)
 thus $k < \text{length } (zs @ [ii])$ by simp
qed moreover
 from $2(5)$ have $\text{length } (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T) + \text{Suc } ii =$
 $\text{length } T + 1$ by (simp add: mk-eqcl'-len) moreover
 from $H \text{ Suc } J$ have $i' < \text{length } (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T) j' <$
 $\text{length } (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T)$ by (simp add: mk-eqcl'-len)+
 ultimately have $IV: (\text{fst } (\text{mk-eqcl}' (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T) (zs$
 $@ [ii]) (\text{Suc } ii) T) ! i' = \text{fst } (\text{mk-eqcl}' (\text{mk-eqcl}' xs ii (\text{Suc } ii) (\text{length } zs) T) (zs @$
 $[ii]) (\text{Suc } ii) T) ! j') =$
 $(\neg \text{tr-lookup } T (i' + \text{Suc } ii) (j' + \text{Suc } ii))$ by (rule $2(1)$)
 with $\text{Suc } J$ show $?thesis$ by (simp add: split-beta)
qed
 } note $L = \text{this}$
 have $i < j \vee i = j \vee i > j$ by auto
 thus $?case$ **proof** (elim disjE)
 assume $i > j$
 with $2(6) L$ have $(\text{fst } (\text{mk-eqcl}' (\text{None} \# xs) zs ii T) ! j = \text{fst } (\text{mk-eqcl}' (\text{None}$
 $\# xs) zs ii T) ! i) = (\neg \text{tr-lookup } T (i + ii) (j + ii))$ by (auto simp: tr-lookup-def)
 thus $?thesis$ by auto
qed (insert $2(7) L$, simp add: tr-lookup-def)+
next
 case $(3 l xs zs ii T i j)$
 { fix $i j$ assume $H: i < j j < \text{length } (\text{Some } l \# xs)$
 then obtain j' where $J: j = \text{Suc } j'$ by (cases j) simp+
 have $(\text{fst } (\text{mk-eqcl}' (\text{Some } l \# xs) zs ii T) ! i = \text{fst } (\text{mk-eqcl}' (\text{Some } l \# xs) zs$
 $ii T) ! j) = (\neg \text{tr-lookup } T (i + ii) (j + ii))$ **proof** (cases i)

case 0
with J **have** $(fst (mk\text{-}eqcl (Some\ l \# xs) zs\ ii\ T) ! i = fst (mk\text{-}eqcl (Some\ l \# xs) zs\ ii\ T) ! j) = (fst (mk\text{-}eqcl\ xs\ zs\ (Suc\ ii)\ T) ! j' = l)$ **by** $(auto\ simp\ add:\ split\text{-}beta)$
also from $3(4)[of\ Some\ l\ l]$ $H\ J$ **have** $\dots = (xs\ !\ j' = Some\ l)$ **by** $(simp\ add:\ mk\text{-}eqcl\text{-}fst\text{-}Some)$
also from J **have** $\dots = ((Some\ l \# xs) ! j = Some\ l)$ **by** $simp$
also from $H\ 0\ 3(2)[of\ i\ j\ l]$ **have** $\dots = (\neg\ tr\text{-}lookup\ T\ (i + ii)\ (j + ii))$ **by** $simp$
finally show $?thesis$ **by** $simp$
next
case $(Suc\ i')$
have $\bigwedge i\ j\ k. \llbracket i < length\ xs; j < length\ xs; xs\ !\ i = Some\ k \rrbracket \implies (xs\ !\ j = Some\ k) = (\neg\ tr\text{-}lookup\ T\ (i + Suc\ ii)\ (j + Suc\ ii))$ **proof** $-$
fix $i\ j\ k$ **assume** $i < length\ xs\ j < length\ xs\ xs\ !\ i = Some\ k$
with $3(2)[of\ Suc\ i\ Suc\ j\ k]$ **show** $(xs\ !\ j = Some\ k) = (\neg\ tr\text{-}lookup\ T\ (i + Suc\ ii)\ (j + Suc\ ii))$ **by** $simp$
qed moreover
note $3(3)$ **moreover**
have $\bigwedge x\ k. \llbracket x \in set\ xs; x = Some\ k \rrbracket \implies k < length\ zs$ **proof** $-$
fix $x\ k$ **assume** $x \in set\ xs\ x = Some\ k$
with $3(4)[of\ x\ k]$ **show** $k < length\ zs$ **by** $simp$
qed moreover
from $3(5)\ H\ Suc\ J$ **have** $length\ xs + Suc\ ii = length\ T + 1\ i' < length\ xs\ j' < length\ xs$ **by** $simp+$
ultimately have $(fst (mk\text{-}eqcl\ xs\ zs\ (Suc\ ii)\ T) ! i' = fst (mk\text{-}eqcl\ xs\ zs\ (Suc\ ii)\ T) ! j') = (\neg\ tr\text{-}lookup\ T\ (i' + Suc\ ii)\ (j' + Suc\ ii))$ **by** $(rule\ 3(1))$
with $J\ Suc$ **show** $?thesis$ **by** $(simp\ add:\ split\text{-}beta)$
qed
} note $L = this$
have $i < j \vee i = j \vee i > j$ **by** $auto$
thus $?case$ **proof** $(elim\ disjE)$
assume $i > j$
with $3(6)\ L$ **have** $(fst (mk\text{-}eqcl (Some\ l \# xs) zs\ ii\ T) ! j = fst (mk\text{-}eqcl (Some\ l \# xs) zs\ ii\ T) ! i) = (\neg\ tr\text{-}lookup\ T\ (j + ii)\ (i + ii))$ **by** $simp$
thus $?thesis$ **by** $(auto\ simp:\ tr\text{-}lookup\text{-}def)$
qed $(insert\ 3(7)\ L,\ simp\ add:\ tr\text{-}lookup\text{-}def)+$
qed

definition $min\text{-}dfa :: dfa \Rightarrow dfa$ **where**

$min\text{-}dfa = (\lambda(bd, as). let (os, ns) = mk\text{-}eqcl (replicate (length\ bd)\ None) []\ 0 (fixpt\ (bd, as)\ (init\text{-}tr\ (bd, as))) in$
 $(map (\lambda p. bdd\text{-}map (\lambda q. os\ !\ q)\ (bd\ !\ p))\ ns,\ map (\lambda p. as\ !\ p)\ ns))$

definition $eq\text{-}nodes :: dfa \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$ **where**

$eq\text{-}nodes = (\lambda M\ v\ p\ q. \neg (\exists n. dist\text{-}nodes\ M\ n\ v\ p\ q))$

lemma $mk\text{-}eqcl\text{-}fixpt\text{-}fst\text{-}bound:$

assumes $dfa\text{-}is\text{-}node\ M\ i$

shows $\text{fst } (\text{mk-} \text{eqcl } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M)))$
 $! \ i < \text{length } (\text{snd } (\text{mk-} \text{eqcl } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))))$
 $(\text{is } \text{fst } ?M \ ! \ i < \text{length } (\text{snd } ?M))$
proof –
{ **fix** $x \ k$ **assume** $H: x \in \text{set } (\text{replicate } (\text{length } (\text{fst } M)) \ (\text{None}::\text{nat } \text{option})) \ x = \text{Some } k$
hence $k < \text{length } \ []$ **by** $(\text{cases } \text{length } (\text{fst } M) = 0) \ \text{simp+}$
} **moreover**
from assms **have** $\text{fst } ?M \ ! \ i \in \text{set } (\text{fst } ?M)$ **by** $(\text{simp } \text{add: } \text{dfa-is-node-def } \text{mk-} \text{eqcl-len-fst})$
ultimately show $?thesis$ **by** $(\text{rule } \text{mk-} \text{eqcl-bound})$
qed

lemma $\text{mk-} \text{eqcl-fixpt-fst-nth}$:

assumes $\text{wf-dfa } M \ v$
and $\text{dfa-is-node } M \ p$ **and** $\text{dfa-is-node } M \ q$
shows $(\text{fst } (\text{mk-} \text{eqcl } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))))$
 $! \ p = \text{fst } (\text{mk-} \text{eqcl } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))) \ ! \ q$
 $= \text{eq-nodes } M \ v \ p \ q$
 $(\text{is } \text{fst } ?M \ ! \ p = \text{fst } ?M \ ! \ q) = \text{eq-nodes } M \ v \ p \ q$
proof –
have $\text{WF: wf-tr } M \ (\text{fixpt } M \ (\text{init-tr } M))$ **by** $(\text{simp } \text{only: } \text{fixpt-wf } \text{init-tr-wf})$

have $(\text{fst } ?M \ ! \ p = \text{fst } ?M \ ! \ q) = (\neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ p \ q)$ **proof**
–
{ **fix** $i \ j \ k$ **assume** $H: i < \text{length } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ j < \text{length } (\text{replicate } (\text{length } (\text{fst } M)) \text{ None}) \ \text{replicate } (\text{length } (\text{fst } M)) \ \text{None} \ ! \ i = \text{Some } k$
hence $(\text{replicate } (\text{length } (\text{fst } M)) \ \text{None} \ ! \ j = \text{Some } k) = (\neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ (i + 0) \ (j + 0))$ **by** simp
}
moreover
have $\bigwedge a \ b \ c. \llbracket a \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)); \ b \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)); \ c \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)); \neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ a \ b; \neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ b \ c \rrbracket$
 $\implies \neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ a \ c$ **proof** –
fix $a \ b \ c$ **assume** $H': a \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)) \ b \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)) \ c \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)) \ \neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ a \ b$
 $\neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ b \ c$
{ **fix** q **assume** $H'': q \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M))$
from assms **have** $\text{length } (\text{fst } M) > 0$ **by** $(\text{simp } \text{add: } \text{wf-dfa-def})$
then obtain m **where** $M: \text{length } (\text{fst } M) = \text{Suc } m$ **by** $(\text{cases } \text{length } (\text{fst } M)) \ \text{simp+}$
hence $M': m = \text{length } (\text{fst } M) - 1$ **by** simp
with $H'' \ \text{WF}$ **have** $q \leq m$ **by** $(\text{simp } \text{add: } \text{wf-tr-def})$
with M **have** $q < \text{length } (\text{fst } M)$ **by** simp
}
with H' **have** $D: \text{dfa-is-node } M \ a \ \text{dfa-is-node } M \ b \ \text{dfa-is-node } M \ c$ **by** $(\text{auto } \text{simp: } \text{dfa-is-node-def})$

with $H'(4,5)$ *assms*(1) **have** $\neg (\exists n. \text{dist-nodes } M \ n \ v \ a \ b) \neg (\exists n. \text{dist-nodes } M \ n \ v \ b \ c)$ **by** (*simp add: fixpt-dist-nodes[symmetric]*)
hence $\neg (\exists n. \text{dist-nodes } M \ n \ v \ a \ c)$ **by** (*auto simp: dist-nodes-def*)
with H' *assms* D **show** $\neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ a \ c$ **by** (*simp add: fixpt-dist-nodes[symmetric]*)
qed
moreover **have** $\bigwedge x \ k. \llbracket x \in \text{set } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}); x = \text{Some } k \rrbracket \implies k < \text{length } []$ **proof** –
fix $x \ k$ **assume** $x \in \text{set } (\text{replicate } (\text{length } (\text{fst } M)) \ (\text{None}::\text{nat option}))$ $x = \text{Some } k$
thus $k < \text{length } []$ **by** (*cases length (fst M) = 0 simp+*)
qed
moreover **from** *WF assms* **have** $\text{length } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) + 0 = \text{length } (\text{fixpt } M \ (\text{init-tr } M)) + 1$ **by** (*simp add: wf-tr-def wf-dfa-def*)
moreover **from** *assms* **have** $p < \text{length } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ q < \text{length } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None})$ **by** (*simp add: dfa-is-node-def*)
ultimately **have** $(\text{fst } ?M \ ! \ p = \text{fst } ?M \ ! \ q) = (\neg \text{tr-lookup } (\text{fixpt } M \ (\text{init-tr } M)) \ (p+0) \ (q+0))$ **by** (*rule mk-egcl-fst-nth*)
thus *?thesis* **by** *simp*
qed
also **from** *assms* **have** $\dots = \text{eq-nodes } M \ v \ p \ q$ **by** (*simp only: fixpt-dist-nodes eq-nodes-def*)
finally **show** *?thesis* **by** *simp*
qed

lemma *mk-egcl-fixpt-fst-snd-nth:*

assumes $i < \text{length } (\text{snd } (\text{mk-egcl } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))))$
and *wf-dfa* $M \ v$
shows $\text{snd } (\text{mk-egcl } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))) \ ! \ i < \text{length } (\text{fst } (\text{mk-egcl } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M)))) \wedge$
 $\text{fst } (\text{mk-egcl } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))) \ ! \ (\text{snd } (\text{mk-egcl } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) \ [] \ 0 \ (\text{fixpt } M \ (\text{init-tr } M))) \ ! \ i) = i$
(is $\text{snd } ?M \ ! \ i < \text{length } (\text{fst } ?M) \wedge \text{fst } ?M \ ! \ (\text{snd } ?M \ ! \ i) = i$ **)**
proof –
have $\bigwedge i. i < \text{length } [] \implies [] \ ! \ i < \text{length } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) + 0$
 $\wedge (0 \leq [] \ ! \ i \longrightarrow \text{replicate } (\text{length } (\text{fst } M)) \ \text{None} \ ! \ ([] \ ! \ i - 0) = \text{Some } i)$ **by** *simp*
moreover **have** $\bigwedge j1 \ j2. \llbracket j1 < j2; j2 < \text{length } [] \rrbracket \implies [] \ ! \ j1 < [] \ ! \ j2$ **by** *simp*
moreover **have** $\bigwedge z. z \in \text{set } [] \implies z < 0$ **by** *simp*
moreover **note** *assms*(1)
moreover **have** $\text{length } (\text{replicate } (\text{length } (\text{fst } M)) \ \text{None}) + 0 \leq \text{length } (\text{fixpt } M \ (\text{init-tr } M)) + 1$ **proof** –
have *WF: wf-tr* $M \ (\text{fixpt } M \ (\text{init-tr } M))$ **by** (*simp only: init-tr-wf fixpt-wf*)
from *assms* **have** $\text{length } (\text{fst } M) > 0$ **by** (*simp add: wf-dfa-def*)
then **obtain** m **where** $M:\text{length } (\text{fst } M) = \text{Suc } m$ **by** (*cases length (fst M)*)
simp+
hence $M': m = \text{length } (\text{fst } M) - 1$ **by** *simp*
with *WF* **have** $\text{length } (\text{fixpt } M \ (\text{init-tr } M)) = m$ **by** (*simp add: wf-tr-def*)

with M **show** $?thesis$ **by** $simp$
qed
ultimately have $snd ?M ! i < length (fst ?M) + 0 \wedge (0 \leq snd ?M ! i \longrightarrow fst ?M ! (snd ?M ! i - 0) = i)$ **by** $(rule\ mk\ eqcl\ fst\ snd)$
thus $?thesis$ **by** $simp$
qed

lemma $eq\ nodes\ dfa\ trans$:
assumes $eq\ nodes\ M\ v\ p\ q$
and $is\ alph\ v\ bs$
shows $eq\ nodes\ M\ v\ (dfa\ trans\ M\ p\ bs)\ (dfa\ trans\ M\ q\ bs)$
proof $(rule\ ccontr)$
assume $H: \neg eq\ nodes\ M\ v\ (dfa\ trans\ M\ p\ bs)\ (dfa\ trans\ M\ q\ bs)$
then obtain $n\ w$ **where** $length\ w = n$ $list\ all\ (is\ alph\ v)\ w$ $dfa\ accepting\ M\ (dfa\ steps\ M\ (dfa\ trans\ M\ p\ bs)\ w) \neq dfa\ accepting\ M\ (dfa\ steps\ M\ (dfa\ trans\ M\ q\ bs)\ w)$
unfolding $eq\ nodes\ def\ dist\ nodes\ def$ **by** $blast$
with $assms$ **have** $length\ (bs\ \# w) = Suc\ n$ $list\ all\ (is\ alph\ v)\ (bs\ \# w)$ $dfa\ accepting\ M\ (dfa\ steps\ M\ p\ (bs\ \# w)) \neq dfa\ accepting\ M\ (dfa\ steps\ M\ q\ (bs\ \# w))$ **by** $simp+$
hence $\neg eq\ nodes\ M\ v\ p\ q$ **unfolding** $eq\ nodes\ def\ dist\ nodes\ def$ **by** $blast$
with $assms$ **show** $False$ **by** $simp$
qed

lemma $mk\ eqcl\ fixpt\ trans$:
assumes $wf\ dfa\ M\ v$
and $dfa\ is\ node\ M\ p$
and $is\ alph\ v\ bs$
shows $dfa\ trans\ (min\ dfa\ M)\ (fst\ (mk\ eqcl\ (replicate\ (length\ (fst\ M))\ None))\ []\ 0\ (fixpt\ M\ (init\ tr\ M)))\ !\ p\ bs =$
 $fst\ (mk\ eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\ tr\ M)))\ !$
 $(dfa\ trans\ M\ p\ bs)$
(is $dfa\ trans\ (min\ dfa\ M)\ (fst\ ?M\ !\ p)\ bs = fst\ ?M\ !\ (dfa\ trans\ M\ p\ bs)$
proof $-$
let $?q = snd\ ?M\ !\ (fst\ ?M\ !\ p)$
from $assms$ **have** $I1: ?q < length\ (fst\ ?M)\ fst\ ?M\ !\ ?q = fst\ ?M\ !\ p$ **by** $(simp\ add:\ mk\ eqcl\ fixpt\ fst\ bound\ mk\ eqcl\ fixpt\ fst\ snd\ nth)+$
with $assms$ **have** $I2: bddh\ (length\ bs)\ (fst\ M\ !\ ?q)$ **by** $(simp\ add:\ mk\ eqcl\ len\ fst\ wf\ dfa\ def\ list\ all\ iff\ is\ alph\ def)$
from $I1$ **have** $I3: dfa\ is\ node\ M\ ?q$ **by** $(simp\ add:\ mk\ eqcl\ len\ fst\ dfa\ is\ node\ def)$
with $assms\ I1$ **have** $eq\ nodes\ M\ v\ p\ ?q$ **by** $(simp\ add:\ mk\ eqcl\ fixpt\ fst\ nth[symmetric])$
with $assms$ **have** $eq\ nodes\ M\ v\ (dfa\ trans\ M\ p\ bs)\ (dfa\ trans\ M\ ?q\ bs)$ **by** $(simp\ add:\ eq\ nodes\ dfa\ trans)$
with $assms\ I3$ **have** $fst\ ?M\ !\ (dfa\ trans\ M\ p\ bs) = fst\ ?M\ !\ (dfa\ trans\ M\ ?q\ bs)$
by $(simp\ add:\ dfa\ trans\ is\ node\ mk\ eqcl\ fixpt\ fst\ nth)$

with $assms\ I2$ **show** $?thesis$ **by** $(simp\ add:\ dfa\ trans\ def\ min\ dfa\ def\ split\ beta\ mk\ eqcl\ fixpt\ fst\ bound\ bdd\ map\ bdd\ lookup)$
qed

lemma *mk-eqcl-fixpt-steps*:
assumes *wf-dfa* M v
and *dfa-is-node* M p
and *list-all* (*is-alph* v) w
shows *dfa-steps* (*min-dfa* M) (*fst* (*mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M))) ! p) w =
fst (*mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M))) !
(*dfa-steps* M p w)
(is *dfa-steps* (*min-dfa* M) (*fst* ? M ! p) w = *fst* ? M ! (*dfa-steps* M p w))
using *assms* **by** (*induct* w *arbitrary*: p) (*simp* *add*: *mk-eqcl-fixpt-trans* *dfa-trans-is-node*)+

lemma *mk-eqcl-fixpt-startnode*:
assumes *length* (*fst* M) > 0
shows *length* (*snd* (*mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M)))) > 0 \wedge
fst (*mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M))) ! 0 =
0 \wedge *snd* (*mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M))) ! 0
= 0
(is *length* (*snd* ? M) > 0 \wedge *fst* ? M ! 0 = 0 \wedge *snd* ? M ! 0 = 0)
proof –
from *assms* **obtain** k **where** K : *length* (*fst* M) = *Suc* k **by** (*cases* *length* (*fst* M)) *simp*+
from K **have** *length* (*snd* ? M) = *length* (*snd* (*mk-eqcl* (*mk-eqcl'* (*replicate* k *None*) 0 (*Suc* 0) 0 (*fixpt* M (*init-tr* M))) [0] (*Suc* 0) (*fixpt* M (*init-tr* M)))) **by**
(*simp* *add*: *split-beta*)
also **have** $\dots \geq$ *length* [0::*nat*] **by** (*simp* *only*: *mk-eqcl-len-snd*)
finally **have** *length* (*snd* ? M) > 0 **by** *auto*
with K **show** ?*thesis* **by** (*simp* *add*: *split-beta* *mk-eqcl-snd-nth*)
qed

lemma *min-dfa-wf*:
wf-dfa M v \implies *wf-dfa* (*min-dfa* M) v
proof –
assume H : *wf-dfa* M v
obtain bd as **where** *min-dfa* M = (bd , as) **by** (*cases* *min-dfa* M) *auto*
hence M : bd = *fst* (*min-dfa* M) as = *snd* (*min-dfa* M) **by** *simp*+
let ? M = *mk-eqcl* (*replicate* (*length* (*fst* M)) *None*) \square 0 (*fixpt* M (*init-tr* M))

{ **fix** x **assume** $x \in$ *set* bd
then **obtain** i **where** I : $i <$ *length* bd x = bd ! i **by** (*auto* *simp*: *in-set-conv-nth*)

with M H **have** *snd* ? M ! $i <$ *length* (*fst* ? M) **by** (*simp* *add*: *min-dfa-def* *split-beta* *mk-eqcl-fixpt-fst-snd-nth*)
hence N : *dfa-is-node* M (*snd* ? M ! i) **by** (*simp* *add*: *mk-eqcl-len-fst* *dfa-is-node-def*)
with H **have** BH : *bddh* v (*fst* M ! (*snd* ? M ! i)) **by** (*simp* *add*: *wf-dfa-def* *list-all-iff* *dfa-is-node-def*)

from I M **have** BI : bd ! i = *bdd-map* ($\lambda q.$ *fst* ? M ! q) (*fst* M ! (*snd* ? M ! i))
by (*simp* *add*: *split-beta* *min-dfa-def*)

with BH **have** $G1$: $bddh\ v\ (bd\ !\ i)$ **by** (*simp add: bddh-bdd-map*)

from $H\ N$ **have** $bdd\ all\ (dfa\ is\ node\ M)\ (fst\ M\ !\ (snd\ ?M\ !\ i))$ **by** (*simp add: wf-dfa-def list-all-iff dfa-is-node-def*)

moreover

{ **fix** q **assume** $dfa\ is\ node\ M\ q$

hence $fst\ ?M\ !\ q < length\ (snd\ ?M)$ **by** (*simp add: mk-eqcl-fixpt-fst-bound*)

hence $dfa\ is\ node\ (min\ dfa\ M)\ (fst\ ?M\ !\ q)$ **by** (*simp add: dfa-is-node-def min-dfa-def split-beta*)

}

ultimately **have** $bdd\ all\ (dfa\ is\ node\ (min\ dfa\ M))\ (bdd\ map\ (\lambda q.\ fst\ ?M\ !\ q)\ (fst\ M\ !\ (snd\ ?M\ !\ i)))$ **by** (*simp add: bdd-all-bdd-map*)

with $G1\ BI\ I$ **have** $bddh\ v\ x \wedge bdd\ all\ (dfa\ is\ node\ (min\ dfa\ M))\ x$ **by** *simp*

}

hence G : $list\ all\ (bddh\ v)\ bd \wedge list\ all\ (bdd\ all\ (dfa\ is\ node\ (min\ dfa\ M)))\ bd$ **by** (*simp add: list-all-iff*)

from H **have** $length\ (fst\ M) > 0$ **by** (*simp add: wf-dfa-def*)

hence $length\ (snd\ ?M) > 0$ **by** (*auto simp only: mk-eqcl-fixpt-startnode*)

with $G\ M$ **show** $wf\ dfa\ (min\ dfa\ M)\ v$ **by** (*simp add: wf-dfa-def min-dfa-def split-beta*)

qed

lemma *min-dfa-accept*:

assumes $wf\ dfa\ M\ v$

and $list\ all\ (is\ alph\ v)\ w$

shows $dfa\ accepts\ (min\ dfa\ M)\ w = dfa\ accepts\ M\ w$

proof –

let $?M = mk\ eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\ tr\ M))$

from *assms* **have** $length\ (fst\ M) > 0$ **by** (*simp add: wf-dfa-def*)

hence SN : $length\ (snd\ ?M) > 0 \wedge fst\ ?M\ !\ 0 = 0 \wedge snd\ ?M\ !\ 0 = 0$ **by** (*auto simp only: mk-eqcl-fixpt-startnode*)

have D : $dfa\ steps\ (min\ dfa\ M)\ 0\ w = fst\ ?M\ !\ dfa\ steps\ M\ 0\ w$ **proof** –

from *assms* **have** $dfa\ is\ node\ M\ 0$ **by** (*simp add: wf-dfa-def dfa-is-node-def*)

moreover **from** SN **have** $dfa\ steps\ (min\ dfa\ M)\ 0\ w = dfa\ steps\ (min\ dfa\ M)\ (fst\ ?M\ !\ 0)\ w$ **by** *simp*

moreover **note** *assms*

ultimately **show** *thesis* **by** (*simp add: mk-eqcl-fixpt-steps*)

qed

from *assms* **have** WF : $wf\ dfa\ (min\ dfa\ M)\ v$ **by** (*simp add: min-dfa-wf*)

hence $dfa\ is\ node\ (min\ dfa\ M)\ 0$ **by** (*simp add: dfa-startnode-is-node*)

with WF *assms* **have** $dfa\ is\ node\ (min\ dfa\ M)\ (dfa\ steps\ (min\ dfa\ M)\ 0\ w)$ **by** (*simp add: dfa-steps-is-node*)

with D **have** DN : $dfa\ is\ node\ (min\ dfa\ M)\ (fst\ ?M\ !\ dfa\ steps\ M\ 0\ w)$ **by** *simp*

let $?q = \text{snd } ?M ! (\text{fst } ?M ! \text{ dfa-steps } M \ 0 \ w)$
from *assms* **have** $N: \text{ dfa-is-node } M \ (\text{ dfa-steps } M \ 0 \ w)$ **by** (*simp add: dfa-steps-is-node dfa-startnode-is-node*)
with *assms* **have** $I: ?q < \text{ length } (\text{fst } ?M)$ $\text{fst } ?M ! ?q = \text{fst } ?M ! \text{ dfa-steps } M \ 0 \ w$ **by** (*simp add: mk-eqcl-fixpt-fst-bound mk-eqcl-fixpt-fst-snd-nth*)
hence $\text{ dfa-is-node } M \ ?q$ **by** (*simp add: mk-eqcl-len-fst dfa-is-node-def*)
with *assms* $N \ I$ **have** $EQ: \text{ eq-nodes } M \ v \ (\text{ dfa-steps } M \ 0 \ w) \ ?q$ **by** (*simp add: mk-eqcl-fixpt-fst-nth[symmetric]*)
have $A: \text{ dfa-accepting } M \ (\text{ dfa-steps } M \ 0 \ w) = \text{ dfa-accepting } M \ ?q$ **proof** (*rule ccontr*)
assume $H: \text{ dfa-accepting } M \ (\text{ dfa-steps } M \ 0 \ w) \neq \text{ dfa-accepting } M \ ?q$
hence $\text{ dist-nodes } M \ 0 \ v \ (\text{ dfa-steps } M \ 0 \ w) \ ?q$ **by** (*auto simp: dist-nodes-def*)
with EQ **show** *False* **by** (*simp add: eq-nodes-def*)
qed

from D **have** $\text{ dfa-accepts } (\text{ min-dfa } M) \ w = \text{ snd } (\text{ min-dfa } M) ! (\text{fst } ?M ! \text{ dfa-steps } M \ 0 \ w)$ **by** (*simp add: accepts-def dfa-accepting-def*)
also from $WF \ DN$ **have** $\dots = \text{ dfa-accepting } M \ ?q$ **by** (*simp add: dfa-is-node-def wf-dfa-def min-dfa-def split-beta dfa-accepting-def*)
also from A **have** $\dots = \text{ dfa-accepts } M \ w$ **by** (*simp add: accepts-def*)

finally show *?thesis* **by** *simp*
qed

4 NFAs

type-synonym $\text{ nbdhtable} = \text{ bool list bdd list}$
type-synonym $\text{ nfa} = \text{ nbdhtable} \times \text{ astate}$

definition

$\text{ nfa-is-node} :: \text{ nfa} \Rightarrow \text{ bool list} \Rightarrow \text{ bool}$ **where**
 $\text{ nfa-is-node } A = (\lambda qs. \text{ length } qs = \text{ length } (\text{fst } A))$

definition

$\text{ wf-nfa} :: \text{ nfa} \Rightarrow \text{ nat} \Rightarrow \text{ bool}$ **where**
 $\text{ wf-nfa } A \ n =$
 $(\text{ list-all } (\text{ bddh } n) (\text{fst } A) \wedge$
 $\text{ list-all } (\text{ bdd-all } (\text{ nfa-is-node } A)) (\text{fst } A) \wedge$
 $\text{ length } (\text{snd } A) = \text{ length } (\text{fst } A) \wedge$
 $\text{ length } (\text{fst } A) > 0)$

definition

$\text{ set-of-bv} :: \text{ bool list} \Rightarrow \text{ nat set}$ **where**
 $\text{ set-of-bv } bs = \{i. i < \text{ length } bs \wedge bs ! i\}$

fun

$\text{ bv-or} :: \text{ bool list} \Rightarrow \text{ bool list} \Rightarrow \text{ bool list}$
where

$bv\text{-or } [] [] = [] \mid$
 $bv\text{-or } (x \# xs) (y \# ys) = (x \vee y) \# (bv\text{-or } xs \ ys)$

lemma *bv-or-nth*:
assumes $length\ l = length\ r$
assumes $i < length\ l$
shows $bv\text{-or } l\ r\ !\ i = (l\ !\ i \vee r\ !\ i)$
using *assms* **proof** (*induct l r arbitrary: i rule: bv-or.induct*)
case ($2\ xx\ xss\ yy\ yss\ ii$)
have $ii = 0 \vee ii > 0$ **by** *auto*
thus *?case* **proof** (*elim disjE*)
assume $ii > 0$
then obtain j **where** $J: ii = Suc\ j$ **by** (*induct ii*) *simp+*
with 2 **show** *?thesis* **by** *simp*
qed *simp*
qed *simp+*

lemma *bv-or-length*:
assumes $length\ l = length\ r$
shows $length\ (bv\text{-or } l\ r) = length\ l$
using *assms* **by** (*induct l r rule: bv-or.induct*) *simp+*

lemma *bv-or-set-of-bv*:
assumes *nfa-is-node* $A\ p$ **and** *nfa-is-node* $A\ q$
shows $set\text{-of}\text{-bv } (bv\text{-or } p\ q) = set\text{-of}\text{-bv } p \cup set\text{-of}\text{-bv } q$
using *assms* **by** (*auto simp: nfa-is-node-def set-of-bv-def bv-or-length bv-or-nth*)

lemma *bv-or-is-node*: $\llbracket nfa\text{-is-node } A\ p; nfa\text{-is-node } A\ q \rrbracket \implies nfa\text{-is-node } A\ (bv\text{-or } p\ q)$
by (*simp add: bv-or-length nfa-is-node-def*)

fun *subsetbdd*
where
 $subsetbdd\ []\ []\ bdd = bdd$
 $\mid subsetbdd\ (bdd'\ \# \ bdds)\ (b\ \# \ bs)\ bdd =$
 $(if\ b\ then\ subsetbdd\ bdds\ bs\ (bdd\text{-binop } bv\text{-or } bdd\ bdd')\ else\ subsetbdd\ bdds\ bs\ bdd)$

definition
 $nfa\text{-emptybdd} :: nat \Rightarrow bool\ list\ bdd$ **where**
 $nfa\text{-emptybdd } n = Leaf\ (replicate\ n\ False)$

lemma *bdd-all-is-node-subsetbdd*:
assumes $list\text{-all } (bdd\text{-all } (nfa\text{-is-node } A))\ (fst\ A)$
and *nfa-is-node* $A\ q$
shows $bdd\text{-all } (nfa\text{-is-node } A)\ (subsetbdd\ (fst\ A)\ q\ (nfa\text{-emptybdd } (length\ q)))$
proof –
 $\{ fix\ bdds :: bool\ list\ bdd\ list\ \mathbf{and}\ q :: bool\ list\ \mathbf{and}\ bd :: bool\ list\ bdd\ \mathbf{and}\ n$
 $\ \ \ \mathbf{assume}\ list\text{-all } (bdd\text{-all } (\lambda l. length\ l = n))\ bdds\ \mathbf{and}\ bdd\text{-all } (\lambda l. length\ l = n)$

bd and $\text{length } bdds = \text{length } q$
hence $\text{bdd-all } (\lambda l. \text{length } l = n) (\text{subsetbdd } bdds \ q \ bd)$ **by** (*induct bdds q bd rule: subsetbdd.induct*) (*simp add: bdd-all-bdd-binop[of $\lambda l. \text{length } l = n - \lambda l. \text{length } l = n$] bv-or-length*)
}
with *assms* **show** *?thesis* **by** (*simp add: nfa-is-node-def nfa-emptybdd-def*)
qed

lemma *bddh-subsetbdd*:
assumes *list-all (bddh l) (fst A)*
and *bddh l bdd'*
and *nfa-is-node A q*
shows *bddh l (subsetbdd (fst A) q bdd')*
using *assms* **unfolding** *nfa-is-node-def* **by** (*induct (fst A) q bdd' rule: subsetbdd.induct*) (*simp add: bddh-binop*)
+

lemma *bdd-lookup-subsetbdd'*:
assumes $\text{length } bdds = \text{length } q$
and $\forall x \in \text{set } bdds. \text{bddh } (\text{length } ws) \ x$
and $\text{bddh } (\text{length } ws) \ \text{obdd}$
and $\bigwedge bs \ w. \llbracket bs \in \text{set } bdds; \text{length } w = \text{length } ws \rrbracket \implies \text{length } (\text{bdd-lookup } bs \ w) = c$
and $\bigwedge w. \text{length } w = \text{length } ws \implies \text{length } (\text{bdd-lookup } \text{obdd } w) = c$
and $a < c$
shows $\text{bdd-lookup } (\text{subsetbdd } bdds \ q \ \text{obdd}) \ ws \ ! \ a = ((\exists i < \text{length } q. q \ ! \ i \wedge \text{bdd-lookup } (bdds \ ! \ i) \ ws \ ! \ a) \vee \text{bdd-lookup } \text{obdd } ws \ ! \ a)$
using *assms* **proof** (*induct bdds q obdd rule: subsetbdd.induct*)
case (2 *bdd' bdds x xs bdd*)
show *?case* **proof** (*cases x*)
case *True*
with 2 **have** *H: bdd-lookup (subsetbdd bdds xs (bdd-binop bv-or bdd bdd')) ws ! a =*
 $((\exists i < \text{length } xs. xs \ ! \ i \wedge \text{bdd-lookup } (bdds \ ! \ i) \ ws \ ! \ a) \vee \text{bdd-lookup } (\text{bdd-binop } bv\text{-or } bdd \ bdd') \ ws \ ! \ a)$ **by** (*simp add: bddh-binop bdd-lookup-binop bv-or-length*)
from 2 **have** $((\exists i < \text{length } xs. xs \ ! \ i \wedge \text{bdd-lookup } (bdds \ ! \ i) \ ws \ ! \ a) \vee \text{bdd-lookup } (\text{bdd-binop } bv\text{-or } bdd \ bdd') \ ws \ ! \ a) =$
 $((\exists i < \text{length } xs. xs \ ! \ i \wedge \text{bdd-lookup } (bdds \ ! \ i) \ ws \ ! \ a) \vee (\text{bdd-lookup } bdd' \ ws) \ ! \ a \vee (\text{bdd-lookup } bdd \ ws) \ ! \ a)$ **by** (*auto simp: bdd-lookup-binop bv-or-nth*)
also **have** $\dots = ((\exists i < \text{Suc } (\text{length } xs). (\text{True} \ \# \ xs) \ ! \ i \wedge \text{bdd-lookup } ((bdd' \ \# \ bdds) \ ! \ i) \ ws \ ! \ a) \vee \text{bdd-lookup } bdd \ ws \ ! \ a)$
(is $((\exists i. ?P \ i) \vee ?Q \vee ?R) = ((\exists i. ?S \ i) \vee ?R)$) **proof**
assume $(\exists i. ?P \ i) \vee ?Q \vee ?R$ **thus** $(\exists i. ?S \ i) \vee ?R$ **by** (*elim disjE*) *auto*
next
assume $(\exists i. ?S \ i) \vee ?R$ **thus** $(\exists i. ?P \ i) \vee ?Q \vee ?R$ **proof** (*elim disjE*)
assume $\exists i. ?S \ i$
then **obtain** *i* **where** *I: ?S i ..*
{ **assume** $i = 0$ **with** *I* **have** *?Q* **by** *simp* **}**
{ **assume** $i \neq 0$ **then** **obtain** *j* **where** $i = \text{Suc } j$ **by** (*cases i*) *simp+* **with** *I* **have** $\exists j. ?P \ j$ **by** *auto* **}**

```

    with ⟨i=0 ⇒ ?Q⟩ show ?thesis by (cases i=0) simp+
  qed simp
qed
finally have ((∃ i < length xs. xs ! i ∧ bdd-lookup (bdds ! i) ws ! a) ∨ bdd-lookup
(bdd-binop bv-or bdd bdd') ws ! a) =
  ((∃ i < Suc (length xs). (True # xs) ! i ∧ bdd-lookup ((bdd' # bdds) ! i) ws !
a) ∨ bdd-lookup bdd ws ! a) by simp
  with True H show ?thesis by simp
next
case False
  with 2 have H: bdd-lookup (subsetbdd bdds xs bdd) ws ! a = ((∃ i < length xs.
xs ! i ∧ bdd-lookup (bdds ! i) ws ! a) ∨ bdd-lookup bdd ws ! a) by simp
  have ((∃ i < length xs. xs ! i ∧ bdd-lookup (bdds ! i) ws ! a) ∨ bdd-lookup bdd ws
! a) =
    ((∃ i < Suc (length xs). (False # xs) ! i ∧ bdd-lookup ((bdd' # bdds) ! i) ws !
a) ∨ bdd-lookup bdd ws ! a)
    (is ((∃ i. ?S i) ∨ ?R) = ((∃ i. ?P i) ∨ ?R)) proof
      assume (∃ i. ?S i) ∨ ?R thus (∃ i. ?P i) ∨ ?R by (elim disjE) auto
    next
      assume (∃ i. ?P i) ∨ ?R thus (∃ i. ?S i) ∨ ?R proof (elim disjE)
        assume ∃ i. ?P i
          then obtain i where ?P i ..
          then obtain j where i = Suc j by (cases i) simp+
          with ⟨?P i⟩ show ?thesis by auto
        qed simp
      qed
    with False H show ?thesis by simp
  qed
qed simp+

```

lemma *bdd-lookup-subsetbdd*:

```

  assumes wf-nfa N (length ws)
  and nfa-is-node N q
  and a < length (fst N)
  shows bdd-lookup (subsetbdd (fst N) q (nfa-emptybdd (length q))) ws ! a = (∃ i <
length q. q ! i ∧ bdd-lookup (fst N ! i) ws ! a)
proof –
  {
    fix w :: bool list
    assume H: length w = length ws
    from assms have ∀ bd ∈ set (fst N). bdd-all (nfa-is-node N) bd by (simp add:
wf-nfa-def list-all-iff) moreover
    from assms have ∀ bd ∈ set (fst N). bddh (length ws) bd by (simp add:
wf-nfa-def list-all-iff) moreover
    note H
    ultimately have ∀ bd ∈ set (fst N). nfa-is-node N (bdd-lookup bd w) by (simp
add: bdd-all-bdd-lookup)
  }
  with assms have bdd-lookup (subsetbdd (fst N) q (nfa-emptybdd (length q)))

```

```

ws ! a = (( $\exists i < \text{length } q. q ! i \wedge \text{bdd-lookup } (\text{fst } N ! i) \text{ ws ! } a$ )  $\vee$   $\text{bdd-lookup } (\text{nfa-emptybdd } (\text{length } q)) \text{ ws ! } a$ )
  by (simp add:  $\text{bdd-lookup-subsetbdd}'$   $\text{nfa-is-node-def}$   $\text{wf-nfa-def}$   $\text{list-all-iff}$   $\text{nfa-emptybdd-def}$ )
  with assms show ?thesis by (auto simp:  $\text{nfa-emptybdd-def}$   $\text{nfa-is-node-def}$ )
qed

```

definition

```

nfa-trans :: nfa  $\Rightarrow$  bool list  $\Rightarrow$  bool list  $\Rightarrow$  bool list where
nfa-trans A qs bs =  $\text{bdd-lookup } (\text{subsetbdd } (\text{fst } A) \text{ qs } (\text{nfa-emptybdd } (\text{length } \text{qs})))$ 
bs

```

```

fun nfa-accepting' :: bool list  $\Rightarrow$  bool list  $\Rightarrow$  bool where

```

```

  nfa-accepting' [] bs = False
| nfa-accepting' as [] = False
| nfa-accepting' (a # as) (b # bs) = (a  $\wedge$  b  $\vee$  nfa-accepting' as bs)

```

```

definition nfa-accepting :: nfa  $\Rightarrow$  bool list  $\Rightarrow$  bool where

```

```

  nfa-accepting A = nfa-accepting' (snd A)

```

```

lemma nfa-accepting'-set-of-bv: nfa-accepting' l r = (set-of-bv l  $\cap$  set-of-bv r  $\neq$  {})

```

```

proof -

```

```

  have nfa-accepting-help:  $\bigwedge as \ q. \text{nfa-accepting}' as \ q = (\exists i. i < \text{length } as \wedge i < \text{length } q \wedge as ! i \wedge q ! i)$ 

```

```

  proof -

```

```

    fix as q

```

```

    show nfa-accepting' as q =  $(\exists i < \text{length } as. i < \text{length } q \wedge as ! i \wedge q ! i)$ 

```

```

    proof (induct as q rule: nfa-accepting'.induct)

```

```

      case (3 a as q qs)

```

```

      thus ?case proof (cases a $\wedge$ q)

```

```

        case False

```

```

        with 3 have nfa-accepting' as qs =  $(\exists i < \text{length } as. i < \text{length } qs \wedge as ! i \wedge qs ! i)$  (is ?T = -) by simp

```

```

        also have ... =  $(\exists j < \text{length } as. j < \text{length } qs \wedge (a\#as) ! \text{Suc } j \wedge (q\#qs) ! \text{Suc } j)$  by simp

```

```

        also have ... =  $(\exists j < \text{length } (a\#as). j < \text{length } (q\#qs) \wedge (a\#as) ! j \wedge (q\#qs) ! j)$  (is  $(\exists j. ?P \ j) = (\exists j. ?Q \ j)$ )

```

```

        proof

```

```

          assume  $\exists j. ?P \ j$ 

```

```

          then obtain j where ?P j ..

```

```

          hence ?Q (Suc j) by simp

```

```

          thus  $\exists j. ?Q \ j$  by (rule exI)

```

```

        next

```

```

          assume  $\exists j. ?Q \ j$ 

```

```

          then obtain j where J: ?Q j ..

```

```

          with False obtain i where j = Suc i by (cases j) simp+

```

```

          with J have ?P i by simp

```

```

          thus  $\exists i. ?P \ i$  by (rule exI)

```

```

        qed

```

```

        also from False have ... =  $((a\wedge q \wedge ?Q \ 0) \vee (\neg (a\wedge q) \wedge (\exists j. ?Q \ j)))$  by

```

auto

also have ... = $((a \wedge q \wedge (\exists j. ?Q j)) \vee (\neg(a \wedge q) \wedge (\exists j. ?Q j)))$ **by** *auto*

also have ... = $(\exists j. ?Q j)$ **by** *auto*

finally have $?T = (\exists j. ?Q j)$.

with *False* **show** *?thesis* **by** *auto*

qed (*auto simp: 3*)

qed *simp+*

qed

hence *nfa-accepting' l r* = $(\exists i. i < \text{length } l \wedge i < \text{length } r \wedge l ! i \wedge r ! i)$ **by** *simp*

also have ... = $(\exists i. i \in \text{set-of-bv } l \wedge i \in \text{set-of-bv } r)$ **by** (*auto simp: set-of-bv-def*)

also have ... = $(\text{set-of-bv } l \cap \text{set-of-bv } r \neq \{\})$ **by** *auto*

finally show *?thesis* .

qed

lemma *nfa-accepting-set-of-bv*: *nfa-accepting A q* = $(\text{set-of-bv } (\text{snd } A) \cap \text{set-of-bv } q \neq \{\})$

by (*simp add: nfa-accepting'-set-of-bv nfa-accepting-def*)

definition

nfa-startnode :: *nfa* \Rightarrow *bool list* **where**

nfa-startnode A = $(\text{replicate } (\text{length } (\text{fst } A)) \text{ False})[0 := \text{True}]$

locale *aut-nfa* =

fixes *A n*

assumes *well-formed: wf-nfa A n*

sublocale *aut-nfa* < *Automaton nfa-trans A nfa-is-node A is-alph n*

proof

fix *q a*

assume *Q: nfa-is-node A q* **and** *A: is-alph n a*

with *well-formed* **have** *bdd-all (nfa-is-node A) (subsetbdd (fst A) q (nfa-emptybdd (length q)))*

by (*simp add: wf-nfa-def bdd-all-is-node-subsetbdd*)

moreover from *well-formed Q* **have** *bddh n (subsetbdd (fst A) q (nfa-emptybdd (length q)))*

by (*simp add: wf-nfa-def nfa-emptybdd-def bddh-subsetbdd*)

with *A* **have** *bddh (length a) (subsetbdd (fst A) q (nfa-emptybdd (length q)))* **by** (*simp add: is-alph-def*)

ultimately have *nfa-is-node A (bdd-lookup (subsetbdd (fst A) q (nfa-emptybdd (length q))) a)*

by (*simp add: bdd-all-bdd-lookup*)

then show *nfa-is-node A (nfa-trans A q a)* **by** (*simp add: nfa-trans-def*)

qed

context *aut-nfa* **begin**

lemmas *trans-is-node* = *trans-is-node*

lemmas *steps-is-node* = *steps-is-node*

lemmas *reach-is-node* = *reach-is-node*

end

lemmas *nfa-trans-is-node* = *aut-nfa.trans-is-node* [*OF aut-nfa.intro*]
lemmas *nfa-steps-is-node* = *aut-nfa.steps-is-node* [*OF aut-nfa.intro*]
lemmas *nfa-reach-is-node* = *aut-nfa.reach-is-node* [*OF aut-nfa.intro*]

abbreviation *nfa-steps* *A* \equiv *foldl* (*nfa-trans* *A*)
abbreviation *nfa-accepts* *A* \equiv *accepts* (*nfa-trans* *A*) (*nfa-accepting* *A*) (*nfa-startnode* *A*)
abbreviation *nfa-reach* *A* \equiv *reach* (*nfa-trans* *A*)

lemma *nfa-startnode-is-node*: *wf-nfa* *A* *n* \implies *nfa-is-node* *A* (*nfa-startnode* *A*)
by (*simp add: nfa-is-node-def wf-nfa-def nfa-startnode-def*)

5 Automata Constructions

5.1 Negation

definition
negate-dfa :: *dfa* \Rightarrow *dfa* **where**
negate-dfa = ($\lambda(t,a).$ (*t*, *map Not a*))

lemma *negate-wf-dfa*: *wf-dfa* (*negate-dfa* *A*) *l* = *wf-dfa* *A* *l*
by (*simp add: negate-dfa-def wf-dfa-def dfa-is-node-def split-beta*)

lemma *negate-negate-dfa*: *negate-dfa* (*negate-dfa* *A*) = *A*
proof (*induct A*)
case (*Pair t a*) **thus** ?*case* **by** (*induct a*) (*simp add: negate-dfa-def*)
qed

lemma *dfa-accepts-negate*:
assumes *wf-dfa* *A* *n*
and *list-all* (*is-alph* *n*) *bss*
shows *dfa-accepts* (*negate-dfa* *A*) *bss* = (\neg *dfa-accepts* *A* *bss*)
proof –
have *dfa-steps* (*negate-dfa* *A*) 0 *bss* = *dfa-steps* *A* 0 *bss*
by (*simp add: negate-dfa-def dfa-trans-def [abs-def] split-beta*)
moreover from *assms* **have** *dfa-is-node* *A* (*dfa-steps* *A* 0 *bss*)
by (*simp add: dfa-steps-is-node dfa-startnode-is-node*)
ultimately show ?*thesis* **using** *assms*
by (*simp add: accepts-def dfa-accepting-def wf-dfa-def dfa-is-node-def negate-dfa-def split-beta*)
qed

5.2 Product Automaton

definition
prod-succs :: *dfa* \Rightarrow *dfa* \Rightarrow *nat* \times *nat* \Rightarrow (*nat* \times *nat*) *list* **where**
prod-succs *A* *B* = ($\lambda(i, j).$ *add-leaves* (*bdd-binop* *Pair* (*fst A* ! *i*) (*fst B* ! *j*)) [])

definition

$prod-is-node\ A\ B = (\lambda(i, j). dfa-is-node\ A\ i \wedge dfa-is-node\ B\ j)$

definition

$prod-invariant :: dfa \Rightarrow dfa \Rightarrow nat\ option\ list\ list \times (nat \times nat)\ list \Rightarrow bool$
where
 $prod-invariant\ A\ B = (\lambda(tab, ps).$
 $length\ tab = length\ (fst\ A) \wedge (\forall\ tab' \in set\ tab. length\ tab' = length\ (fst\ B)))$

definition

$prod-ins = (\lambda(i, j). \lambda(tab, ps).$
 $(tab[i := (tab ! i)]j := Some\ (length\ ps)]],$
 $ps\ @\ [(i, j)])$

definition

$prod-memb :: nat \times nat \Rightarrow nat\ option\ list\ list \times (nat \times nat)\ list \Rightarrow bool$ **where**
 $prod-memb = (\lambda(i, j). \lambda(tab, ps). tab ! i ! j \neq None)$

definition

$prod-empt :: dfa \Rightarrow dfa \Rightarrow nat\ option\ list\ list \times (nat \times nat)\ list$ **where**
 $prod-empt\ A\ B = (replicate\ (length\ (fst\ A))\ (replicate\ (length\ (fst\ B))\ None),\ [])$

definition

$prod-dfs :: dfa \Rightarrow dfa \Rightarrow nat \times nat \Rightarrow nat\ option\ list\ list \times (nat \times nat)\ list$
where
 $prod-dfs\ A\ B\ x = gen-dfs\ (prod-succs\ A\ B)\ prod-ins\ prod-memb\ (prod-empt\ A\ B)$
 $[x]$

definition

$binop-dfa :: (bool \Rightarrow bool \Rightarrow bool) \Rightarrow dfa \Rightarrow dfa \Rightarrow dfa$ **where**
 $binop-dfa\ f\ A\ B =$
 $(let\ (tab, ps) = prod-dfs\ A\ B\ (0, 0)$
 in
 $(map\ (\lambda(i, j). bdd-binop\ (\lambda k\ l. the\ (tab ! k ! l))\ (fst\ A ! i)\ (fst\ B ! j))\ ps,$
 $map\ (\lambda(i, j). f\ (snd\ A ! i)\ (snd\ B ! j))\ ps))$

locale $prod-DFS =$

fixes $A\ B\ n$
assumes $well-formed1: wf-dfa\ A\ n$
and $well-formed2: wf-dfa\ B\ n$

sublocale $prod-DFS < DFS\ prod-succs\ A\ B\ prod-is-node\ A\ B\ prod-invariant\ A\ B$
 $prod-ins\ prod-memb\ prod-empt\ A\ B$

apply $unfold-locales$
apply $(simp\ add: prod-memb-def\ prod-ins-def\ prod-invariant-def$
 $prod-is-node-def\ split-paired-all\ dfa-is-node-def)$
apply $(case-tac\ a = aa)$
apply $(case-tac\ b = ba)$

```

apply auto[3]
apply (simp add: prod-memb-def prod-empt-def prod-is-node-def split-paired-all
dfa-is-node-def)
apply (insert well-formed1 well-formed2)[]
apply (simp add: prod-is-node-def prod-succs-def split-paired-all dfa-is-node-def
wf-dfa-def)
apply (drule conjunct1 [OF conjunct2])+
apply (simp add: list-all-iff)
apply (rule ballI)
apply (simp add: split-paired-all)
apply (drule subsetD [OF add-leaves-binop-subset [where xs=[] and ys=[] , sim-
plified]])
apply clarify
apply (drule-tac x=fst A ! a in bspec)
apply simp
apply (drule-tac x=fst B ! b in bspec)
apply simp
apply (simp add: add-leaves-bdd-all-eq' list-all-iff)
apply (simp add: prod-invariant-def prod-empt-def set-replicate-conv-if)
apply (simp add: prod-is-node-def prod-invariant-def
prod-memb-def prod-ins-def split-paired-all dfa-is-node-def)
apply (rule ballI)
apply (drule subsetD [OF set-update-subset-insert])
apply auto
apply (simp add: prod-is-node-def dfa-is-node-def)
done

```

```

context prod-DFS
begin

```

```

lemma prod-dfs-eq-rtrancl: prod-is-node A B x  $\implies$  prod-is-node A B y  $\implies$ 
prod-memb y (prod-dfs A B x) = ((x, y)  $\in$  (succsr (prod-succs A B))*
by (unfold prod-dfs-def) (rule dfs-eq-rtrancl)

```

```

lemma prod-dfs-bij:

```

```

assumes x: prod-is-node A B x
shows (fst (prod-dfs A B x) ! i ! j = Some k  $\wedge$  dfa-is-node A i  $\wedge$  dfa-is-node B
j) =
(k < length (snd (prod-dfs A B x))  $\wedge$  (snd (prod-dfs A B x) ! k = (i, j)))
proof –
from x have list-all (prod-is-node A B) [x] by simp
with empt-invariant
have (fst (dfs (prod-empt A B) [x]) ! i ! j = Some k  $\wedge$  dfa-is-node A i  $\wedge$  dfa-is-node
B j) =
(k < length (snd (dfs (prod-empt A B) [x]))  $\wedge$  (snd (dfs (prod-empt A B) [x]) !
k = (i, j)))
proof (induct rule: dfs-invariant)
case base
show ?case

```

```

    by (auto simp add: prod-empt-def dfa-is-node-def)
next
case (step S y)
obtain y1 y2 where y: y = (y1, y2) by (cases y)
show ?case
proof (cases y1 = i)
  case True
  show ?thesis
  proof (cases y2 = j)
    case True
    with step y ⟨y1 = i⟩ show ?thesis
    by (auto simp add: prod-ins-def prod-memb-def split-beta nth-append
        prod-invariant-def prod-is-node-def dfa-is-node-def)
  next
  case False
  with step y ⟨y1 = i⟩ show ?thesis
  by (auto simp add: prod-ins-def prod-memb-def split-beta nth-append
      prod-invariant-def prod-is-node-def dfa-is-node-def)
  qed
next
case False
with step y show ?thesis
by (auto simp add: prod-ins-def prod-memb-def split-beta nth-append)
qed
qed
then show ?thesis by (simp add: prod-dfs-def)
qed

lemma prod-dfs-mono:
  assumes z: prod-invariant A B z
  and xs: list-all (prod-is-node A B) xs
  and H: fst z ! i ! j = Some k
  shows fst (gen-dfs (prod-succs A B) prod-ins prod-memb z xs) ! i ! j = Some k
  using z xs
  apply (rule dfs-invariant)
  apply (rule H)
  apply (simp add: prod-ins-def prod-memb-def split-paired-all prod-is-node-def
    prod-invariant-def)
  apply (case-tac aa = i)
  apply (case-tac ba = j)
  apply (simp add: dfa-is-node-def)+
  done

lemma prod-dfs-start:
  [[dfa-is-node A i; dfa-is-node B j]]  $\implies$  fst (prod-dfs A B (i, j)) ! i ! j = Some 0
  apply (simp add: prod-dfs-def empty prod-is-node-def gen-dfs-simps)
  apply (rule prod-dfs-mono)
  apply (rule ins-invariant)
  apply (simp add: prod-is-node-def dfa-is-node-def)

```



```

apply (rule empt-invariant)
apply (rule empt)
apply (simp add: prod-is-node-def)
apply (rule succs-is-node)
apply (simp add: prod-is-node-def)
apply (simp add: prod-ins-def prod-empt-def dfa-is-node-def)
done

```

lemma *prod-dfs-inj*:

assumes x : *prod-is-node* $A B x$ **and** $i1$: *dfa-is-node* $A i1$ **and** $i2$: *dfa-is-node* $B i2$

and $j1$: *dfa-is-node* $A j1$ **and** $j2$: *dfa-is-node* $B j2$

and i : *fst* (*prod-dfs* $A B x$) ! $i1$! $i2$ = *Some* k

and j : *fst* (*prod-dfs* $A B x$) ! $j1$! $j2$ = *Some* k

shows $(i1, i2) = (j1, j2)$

proof –

from $x i1 i2 i$

have $k < \text{length} (\text{snd} (\text{prod-dfs } A B x)) \wedge \text{snd} (\text{prod-dfs } A B x) ! k = (i1, i2)$

by (simp add: *prod-dfs-bij [symmetric]*)

moreover from $x j1 j2 j$

have $k < \text{length} (\text{snd} (\text{prod-dfs } A B x)) \wedge \text{snd} (\text{prod-dfs } A B x) ! k = (j1, j2)$

by (simp add: *prod-dfs-bij [symmetric]*)

ultimately show *?thesis* **by** *simp*

qed

lemma *prod-dfs-statetrans*:

assumes bs : *length* $bs = n$

and i : *dfa-is-node* $A i$ **and** j : *dfa-is-node* $B j$

and $s1$: *dfa-is-node* $A s1$ **and** $s2$: *dfa-is-node* $B s2$

and k : *fst* (*prod-dfs* $A B (s1, s2)$) ! i ! j = *Some* k

obtains k'

where *fst* (*prod-dfs* $A B (s1, s2)$) !

dfa-trans $A i bs$! *dfa-trans* $B j bs$ = *Some* k'

and *dfa-is-node* $A (\text{dfa-trans } A i bs)$

and *dfa-is-node* $B (\text{dfa-trans } B j bs)$

and $k' < \text{length} (\text{snd} (\text{prod-dfs } A B (s1, s2)))$

proof –

from i *well-formed1* bs **have** $h\text{-tr1}$: *bddh* (*length* bs) (*fst* A ! i) **by** (simp add: *wf-dfa-def dfa-is-node-def list-all-iff*)

from j *well-formed2* bs **have** $h\text{-tr2}$: *bddh* (*length* bs) (*fst* B ! j) **by** (simp add: *wf-dfa-def dfa-is-node-def list-all-iff*)

from $i j k$ **have** *prod-memb* (i, j) (*prod-dfs* $A B (s1, s2)$)

by (simp add: *prod-memb-def split-beta*)

then have $((s1, s2), (i, j)) \in (\text{succsr } (\text{prod-succs } A B))^*$

using $i j s1 s2$

by (simp add: *prod-dfs-eq-rtrancl prod-is-node-def*)

moreover from $h\text{-tr1}$ $h\text{-tr2}$ **have** (*bdd-lookup* (*fst* A ! i) bs , *bdd-lookup* (*fst* B ! j) bs) =

bdd-lookup (*bdd-binop* *Pair* (*fst* A ! i) (*fst* B ! j)) bs

```

  by (simp add: bdd-lookup-binop)
with i j h-tr1 h-tr2
have ((i, j), (bdd-lookup (fst A ! i) bs, bdd-lookup (fst B ! j) bs)) ∈
  succsr (prod-succs A B)
  by (auto simp add: succsr-def prod-succs-def
    add-leaves-bdd-lookup [of length bs] bddh-binop is-alph-def)
ultimately have ((s1, s2), (bdd-lookup (fst A ! i) bs, bdd-lookup (fst B ! j) bs))
∈
  (succsr (prod-succs A B))* ..
moreover from well-formed1 well-formed2 bs i j
have prod-is-node A B (bdd-lookup (fst A ! i) bs, bdd-lookup (fst B ! j) bs)
  by (auto simp: prod-is-node-def bdd-all-bdd-lookup is-alph-def dfa-trans-is-node
dfa-trans-def[symmetric])
moreover from i well-formed1 bs
have s-tr1: dfa-is-node A (dfa-trans A i bs)
  by (simp add: is-alph-def dfa-trans-is-node)
moreover from j well-formed2 bs
have s-tr2: dfa-is-node B (dfa-trans B j bs)
  by (simp add: is-alph-def dfa-trans-is-node)
ultimately have ∃ k'. fst (prod-dfs A B (s1, s2)) !
  dfa-trans A i bs ! dfa-trans B j bs = Some k'
  using s1 s2
  by (simp add: prod-dfs-eq-rtrancl [symmetric] prod-memb-def split-beta prod-is-node-def
dfa-trans-def)
then obtain k' where k': fst (prod-dfs A B (s1, s2)) !
  dfa-trans A i bs ! dfa-trans B j bs = Some k' ..
from k' s-tr1 s-tr2 s1 s2
have k' < length (snd (prod-dfs A B (s1, s2))) ∧
  snd (prod-dfs A B (s1, s2)) ! k' = (dfa-trans A i bs, dfa-trans B j bs)
  by (simp add: prod-dfs-bij [symmetric] prod-is-node-def)
then have k' < length (snd (prod-dfs A B (s1, s2))) by simp
with k' s-tr1 s-tr2 show ?thesis ..
qed

```

lemma *binop-wf-dfa*: *wf-dfa (binop-dfa f A B) n*

proof –

```

  let ?dfa = binop-dfa f A B
  from well-formed1 well-formed2 have is-node-s1-s2: prod-is-node A B (0, 0) by
(simp add: prod-is-node-def wf-dfa-def dfa-is-node-def)
  let ?tr = map (λ(i,j). bdd-binop (λk l. the (fst (prod-dfs A B (0, 0)) ! k ! l)) (fst
A ! i) (fst B ! j)) (snd (prod-dfs A B (0,0)))
  {
    fix i j
    assume ij: (i, j) ∈ set (snd (prod-dfs A B (0, 0)))
    then obtain k where k: k < length (snd (prod-dfs A B (0, 0)))
      snd (prod-dfs A B (0, 0)) ! k = (i, j)
      by (auto simp add: in-set-conv-nth)
    from conjI [OF k] obtain ij-k: fst (prod-dfs A B (0,0)) ! i ! j = Some k
      and i: dfa-is-node A i and j: dfa-is-node B j

```

by (simp add: prod-dfs-bij [OF is-node-s1-s2, symmetric])
 from well-formed1 i have bddh-tr1: bddh n (fst A ! i)
 and less-tr1: bdd-all (dfa-is-node A) (fst A ! i) by (simp add: wf-dfa-def
 list-all-iff dfa-is-node-def)+
 from well-formed2 j have bddh-tr2: bddh n (fst B ! j)
 and less-tr2: bdd-all (dfa-is-node B) (fst B ! j) by (simp add: wf-dfa-def
 list-all-iff dfa-is-node-def)+
 from bddh-tr1 bddh-tr2 have 1: bddh n (bdd-binop ($\lambda k l$. the (fst (prod-dfs A B
 (0, 0)) ! k ! l)) (fst A ! i) (fst B ! j))
 by (simp add: bddh-binop)
 have $\forall bs$. length bs = n \longrightarrow the (fst (prod-dfs A B (0, 0)) ! dfa-trans A i bs !
 dfa-trans B j bs)
 < length (snd (prod-dfs A B (0, 0)))
 proof (intro strip)
 fix bs
 assume bs: length (bs::bool list) = n
 moreover note i j
 moreover from well-formed1 well-formed2 have dfa-is-node A 0 and
 dfa-is-node B 0
 by (simp add: dfa-is-node-def wf-dfa-def)+
 moreover note ij-k
 ultimately obtain m where fst (prod-dfs A B (0, 0)) ! dfa-trans A i bs !
 dfa-trans B j bs = Some m
 and m < length (snd (prod-dfs A B (0, 0))) by (rule prod-dfs-statetrans)
 then show the (fst (prod-dfs A B (0,0)) ! dfa-trans A i bs ! dfa-trans B j bs)
 < length (snd (prod-dfs A B (0,0))) by simp
 qed
 with bddh-tr1 bddh-tr2 have 2: bdd-all (λq . q < length (snd (prod-dfs A B (0,
 0)))) (bdd-binop ($\lambda k l$. the (fst (prod-dfs A B (0,0)) ! k ! l)) (fst A ! i) (fst B ! j))
 by (simp add: bddh-binop bdd-lookup-binop bdd-all-bdd-lookup-iff[of n - λx . x
 < length (snd (prod-dfs A B (0,0)))] dfa-trans-def)
 note this 1
 }
 hence 1: list-all (bddh n) ?tr and 2: list-all (bdd-all (λq . q < length ?tr)) ?tr
 by (auto simp: split-paired-all list-all-iff)
 from well-formed1 well-formed2 have 3: fst (prod-dfs A B (0, 0)) ! 0 ! 0 = Some
 0 by (simp add: wf-dfa-def dfa-is-node-def prod-dfs-start)
 from is-node-s1-s2 have (fst (prod-dfs A B (0,0)) ! 0 ! 0 = Some 0 \wedge dfa-is-node
 A 0 \wedge dfa-is-node B 0) =
 (0 < length (snd (prod-dfs A B (0,0))) \wedge snd (prod-dfs A B (0,0)) ! 0 = (0,0))
 by (rule prod-dfs-bij)
 with 3 well-formed1 well-formed2 have 0 < length (snd (prod-dfs A B (0,0)))
 by (simp add: wf-dfa-def dfa-is-node-def)
 with 1 2 3 show wf-dfa (binop-dfa f A B) n by (simp add: binop-dfa-def wf-dfa-def
 split-beta dfa-is-node-def)
 qed

theorem binop-dfa-reachable:
 assumes bss: list-all (is-alph n) bss

shows $(\exists m. \text{dfa-reach } (\text{binop-dfa } f \ A \ B) \ 0 \ \text{bss } m \wedge$
 $\text{fst } (\text{prod-dfs } A \ B \ (0, 0)) \ ! \ s_1 \ ! \ s_2 = \text{Some } m \wedge$
 $\text{dfa-is-node } A \ s_1 \wedge \text{dfa-is-node } B \ s_2) =$
 $(\text{dfa-reach } A \ 0 \ \text{bss } s_1 \wedge \text{dfa-reach } B \ 0 \ \text{bss } s_2)$

proof –

let $?tr = \text{map } (\lambda(i, j).$
 $\text{bdd-binop } (\lambda k \ l. \text{the } (\text{fst } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ k \ ! \ l)) \ (\text{fst } A \ ! \ i) \ (\text{fst } B \ ! \ j))$
 $(\text{snd } (\text{prod-dfs } A \ B \ (0,0)))$

have $T: ?tr = \text{fst } (\text{binop-dfa } f \ A \ B)$ **by** $(\text{simp add: binop-dfa-def split-beta})$

from $\text{well-formed1 well-formed2}$ **have** $\text{is-node-s1-s2: prod-is-node } A \ B \ (0, 0)$ **by**
 $(\text{simp add: prod-is-node-def wf-dfa-def dfa-is-node-def})$

from $\text{well-formed1 well-formed2}$ **have** $s1: \text{dfa-is-node } A \ 0$ **and** $s2: \text{dfa-is-node } B$
 0 **by** $(\text{simp add: dfa-is-node-def wf-dfa-def})+$

from $s1 \ s2$ **have** $\text{start: fst } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ 0 \ ! \ 0 = \text{Some } 0$
by $(\text{rule prod-dfs-start})$

show $(\exists m. \text{dfa-reach } (\text{binop-dfa } f \ A \ B) \ 0 \ \text{bss } m \wedge$
 $\text{fst } (\text{prod-dfs } A \ B \ (0, 0)) \ ! \ s_1 \ ! \ s_2 = \text{Some } m \wedge$
 $\text{dfa-is-node } A \ s_1 \wedge \text{dfa-is-node } B \ s_2) =$
 $(\text{dfa-reach } A \ 0 \ \text{bss } s_1 \wedge \text{dfa-reach } B \ 0 \ \text{bss } s_2)$
 $(\text{is } (\exists m. ?lhs1 \ m \wedge ?lhs2 \ m \wedge ?lhs3 \wedge ?lhs4) = ?rhs$
 $\text{is } ?lhs = -)$

proof

assume $\exists m. ?lhs1 \ m \wedge ?lhs2 \ m \wedge ?lhs3 \wedge ?lhs4$

then obtain m **where** $\text{lhs: } ?lhs1 \ m \ ?lhs2 \ m \ ?lhs3 \ ?lhs4$ **by** auto

from lhs bss **show** $?rhs$

proof $(\text{induct arbitrary: } s_1 \ s_2)$

case Nil

from is-node-s1-s2

$s1 \ s2 \langle \text{dfa-is-node } A \ s_1 \rangle \langle \text{dfa-is-node } B \ s_2 \rangle$

have $(0, 0) = (s_1, s_2)$

using $\text{start } \langle \text{fst } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ s_1 \ ! \ s_2 = \text{Some } 0 \rangle$

by $(\text{rule prod-dfs-inj})$

moreover **have** $\text{dfa-reach } A \ 0 \ [] \ 0$ **by** (rule reach-nil)

moreover **have** $\text{dfa-reach } B \ 0 \ [] \ 0$ **by** (rule reach-nil)

ultimately show $?case$ **by** simp

next

case $(\text{snoc } j \ \text{bss } bs \ s_1 \ s_2)$

then **have** $\text{length } bs = n$ **by** $(\text{simp add: is-alph-def})$

moreover **from** binop-wf-dfa **have** $\text{dfa-is-node } (\text{binop-dfa } f \ A \ B) \ 0$ **by** $(\text{simp$
 $\text{add: dfa-is-node-def wf-dfa-def})$

with $\text{snoc binop-wf-dfa } [of \ f]$ **have** $\text{dfa-is-node } (\text{binop-dfa } f \ A \ B) \ j$ **by** $(\text{simp$
 $\text{add: dfa-reach-is-node})$

then **have** $j: j < \text{length } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)))$ **by** $(\text{simp add: binop-dfa-def$
 $\text{dfa-is-node-def split-beta})$

with $\text{prod-dfs-bij } [OF \ \text{is-node-s1-s2},$
 $\text{of } \text{fst } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ j) \ \text{snd } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ j)]$

have $j\text{-tr1: dfa-is-node } A \ (\text{fst } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ j))$

and $j\text{-tr2: dfa-is-node } B \ (\text{snd } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ j))$

and $\text{Some-j: fst } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ \text{fst } (\text{snd } (\text{prod-dfs } A \ B \ (0,0)) \ ! \ j) \ !$

```

snd (snd (prod-dfs A B (0,0)) ! j) = Some j
  by auto
  note j-tr1 j-tr2 s1 s2 Some-j
  ultimately obtain k
    where k: fst (prod-dfs A B (0,0)) !
      dfa-trans A (fst (snd (prod-dfs A B (0, 0)) ! j)) bs !
      dfa-trans B (snd (snd (prod-dfs A B (0, 0)) ! j)) bs = Some k
    and s-tr1': dfa-is-node A (dfa-trans A (fst (snd (prod-dfs A B (0,0)) ! j)) bs)
    and s-tr2': dfa-is-node B (dfa-trans B (snd (snd (prod-dfs A B (0,0)) ! j))
bs)
      by (rule prod-dfs-statetrans)

from well-formed1 well-formed2 j-tr1 j-tr2 snoc
have lh: bddh (length bs) (fst A ! fst (snd (prod-dfs A B (0,0)) ! j))
  and rh: bddh (length bs) (fst B ! snd (snd (prod-dfs A B (0,0)) ! j))
  by (auto simp: wf-dfa-def dfa-is-node-def list-all-iff is-alph-def)
  from snoc(3)[unfolded dfa-trans-def binop-dfa-def Let-def split-beta fst-conv
nth-map[OF j] bdd-lookup-binop[OF lh,OF rh], folded dfa-trans-def] k
  have fst (prod-dfs A B (0,0)) ! s1 ! s2 = Some k by simp
  with is-node-s1-s2 dfa-is-node A s1 dfa-is-node B s2 s-tr1' s-tr2'
  have (s1, s2) = (dfa-trans A (fst (snd (prod-dfs A B (0,0)) ! j)) bs, dfa-trans
B (snd (snd (prod-dfs A B (0,0)) ! j)) bs)
    using k
    by (rule prod-dfs-inj)
  moreover from snoc Some-j j-tr1 j-tr2
  have dfa-reach A 0 bss (fst (snd (prod-dfs A B (0,0)) ! j)) by simp
  hence dfa-reach A 0 (bss @ [bs]) (dfa-trans A (fst (snd (prod-dfs A B (0,0))
! j)) bs)
    by (rule reach-snoc)
  moreover from snoc Some-j j-tr1 j-tr2
  have dfa-reach B 0 bss (snd (snd (prod-dfs A B (0,0)) ! j)) by simp
  hence dfa-reach B 0 (bss @ [bs]) (dfa-trans B (snd (snd (prod-dfs A B (0,0))
! j)) bs)
    by (rule reach-snoc)
  ultimately show dfa-reach A 0 (bss @ [bs]) s1  $\wedge$  dfa-reach B 0 (bss @ [bs])
s2
    by simp
  qed
next
  assume ?rhs
  hence reach: dfa-reach A 0 bss s1 dfa-reach B 0 bss s2 by simp-all
  then show ?lhs using bss
  proof (induct arbitrary: s2)
    case Nil
      with start s1 s2 show ?case
        by (auto intro: reach-nil simp: reach-nil-iff)
    next
      case (snoc j bss bs s2)
      from snoc(3)

```

```

obtain  $s_2'$  where  $reach-s_2'$ :  $dfa-reach\ B\ 0\ bss\ s_2'$  and  $s_2'$ :  $s_2 = dfa-trans\ B\ s_2'\ bs$ 
  by (auto simp: reach-snoc-iff)
from  $snoc(2)$  [OF reach-s2']  $snoc(4)$ 
obtain  $m$  where  $reach-m$ :  $dfa-reach\ (binop-dfa\ f\ A\ B)\ 0\ bss\ m$ 
  and  $m$ :  $fst\ (prod-dfs\ A\ B\ (0,0))\ !\ j\ !\ s_2' = Some\ m$ 
  and  $j$ :  $dfa-is-node\ A\ j$  and  $s_2''$ :  $dfa-is-node\ B\ s_2''$ 
  by auto
from  $snoc$  have  $list-all\ (is-alph\ n)\ bss$  by simp
with  $binop-wf-dfa\ reach-m\ dfa-startnode-is-node$  [OF binop-wf-dfa]
have  $m-less$ :  $dfa-is-node\ (binop-dfa\ f\ A\ B)\ m$ 
  by (rule dfa-reach-is-node)
from  $is-node-s1-s2\ m\ j\ s_2''$ 
have  $m'$ :  $(m < length\ (snd\ (prod-dfs\ A\ B\ (0,0)))) \wedge$ 
   $snd\ (prod-dfs\ A\ B\ (0,0))\ !\ m = (j,\ s_2')$ 
  by (simp add: prod-dfs-bij [symmetric])
with  $j\ s_2''$  have  $dfa-is-node\ A\ (fst\ (snd\ (prod-dfs\ A\ B\ (0,0))\ !\ m))$ 
   $dfa-is-node\ B\ (snd\ (snd\ (prod-dfs\ A\ B\ (0,0))\ !\ m))$ 
  by simp-all
with  $well-formed1\ well-formed2\ snoc$ 
have  $bddh$ :  $bddh\ (length\ bs)\ (fst\ A\ !\ fst\ (snd\ (prod-dfs\ A\ B\ (0,0))\ !\ m))$ 
   $bddh\ (length\ bs)\ (fst\ B\ !\ snd\ (snd\ (prod-dfs\ A\ B\ (0,0))\ !\ m))$ 
  by (simp add: wf-dfa-def is-alph-def dfa-is-node-def list-all-iff)+
from  $snoc$  have  $length\ bs = n$  by (simp add: is-alph-def)
then obtain  $k$  where  $k$ :  $fst\ (prod-dfs\ A\ B\ (0,0))\ !$ 
   $dfa-trans\ A\ j\ bs\ !\ dfa-trans\ B\ s_2'\ bs = Some\ k$ 
  and  $s-tr1$ :  $dfa-is-node\ A\ (dfa-trans\ A\ j\ bs)$ 
  and  $s-tr2$ :  $dfa-is-node\ B\ (dfa-trans\ B\ s_2'\ bs)$ 
  using  $j\ s_2''\ s1\ s2\ m$ 
  by (rule prod-dfs-statetrans)
show ?case
  apply (rule exI)
  apply (simp add: s2')
  apply (intro conjI)
  apply (rule reach-snoc)
  apply (rule reach-m)
  apply (cut-tac m-less)
  apply (simp add: dfa-trans-def binop-dfa-def split-beta dfa-is-node-def)
  apply (simp add: bddh bdd-lookup-binop split-beta)
  apply (simp add: dfa-trans-def [symmetric] m' k)
  apply (rule s-tr1)
  apply (rule s-tr2)
  done
qed
qed
qed

```

lemma *binop-dfa-steps*:
assumes X : $list-all\ (is-alph\ n)\ bs$

shows $\text{snd} (\text{binop-dfa } f \ A \ B) ! \text{dfa-steps} (\text{binop-dfa } f \ A \ B) \ 0 \ bs = f (\text{snd } A ! \text{dfa-steps } A \ 0 \ bs) (\text{snd } B ! \text{dfa-steps } B \ 0 \ bs)$
(is $?as3 ! \text{dfa-steps } ?A \ 0 \ bs = ?rhs$)
proof –
note $2 = \text{dfa-startnode-is-node}[OF \ \text{well-formed1}]$
note $5 = \text{dfa-startnode-is-node}[OF \ \text{well-formed2}]$
note $B = \text{dfa-startnode-is-node}[OF \ \text{binop-wf-dfa}]$
define tab **where** $tab = \text{fst} (\text{prod-dfs } A \ B \ (0,0))$
define ps **where** $ps = \text{snd} (\text{prod-dfs } A \ B \ (0,0))$
from $tab\text{-def}$ $ps\text{-def}$ **have** $\text{prod: prod-dfs } A \ B \ (0,0) = (tab, ps)$ **by** simp
define $s1$ **where** $s1 = \text{dfa-steps } A \ 0 \ bs$
define $s2$ **where** $s2 = \text{dfa-steps } B \ 0 \ bs$
with $s1\text{-def}$ **have** $\text{dfa-reach } A \ 0 \ bs \ s1$ **and** $\text{dfa-reach } B \ 0 \ bs \ s2$ **by** $(\text{simp add: reach-def})+$
with X **have** $\exists m. \text{dfa-reach } ?A \ 0 \ bs \ m \wedge \text{fst} (\text{prod-dfs } A \ B \ (0, 0)) ! s1 ! s2 = \text{Some } m \wedge \text{dfa-is-node } A \ s1 \wedge \text{dfa-is-node } B \ s2$
by $(\text{simp add: binop-dfa-reachable})$
with $tab\text{-def}$ **have** $\exists m. \text{dfa-reach } ?A \ 0 \ bs \ m \wedge tab ! s1 ! s2 = \text{Some } m \wedge \text{dfa-is-node } A \ s1 \wedge \text{dfa-is-node } B \ s2$ **by** simp
then obtain m **where** $R: \text{dfa-reach } ?A \ 0 \ bs \ m$ **and** $M: tab ! s1 ! s2 = \text{Some } m$
and $s1: \text{dfa-is-node } A \ s1$ **and** $s2: \text{dfa-is-node } B \ s2$ **by** blast
hence $M': m = \text{dfa-steps } ?A \ 0 \ bs$ **by** $(\text{simp add: reach-def})$
from $B \ X \ R \ \text{binop-wf-dfa} \ [of \ f]$ **have** $mL: \text{dfa-is-node } ?A \ m$ **by** $(\text{simp add: dfa-reach-is-node})$
from $2 \ 5 \ M \ s1 \ s2$ **have** $\text{bij: } m < \text{length} (\text{snd} (\text{prod-dfs } A \ B \ (0, 0))) \wedge \text{snd} (\text{prod-dfs } A \ B \ (0, 0)) ! m = (s1, s2)$ **unfolding** $tab\text{-def}$
by $(\text{simp add: prod-dfs-bij[symmetric] prod-is-node-def})$
with mL **have** $\text{snd} (\text{binop-dfa } f \ A \ B) ! m = f (\text{snd } A ! s1) (\text{snd } B ! s2)$
by $(\text{simp add: binop-dfa-def split-beta dfa-is-node-def})$
with $M' \ s1\text{-def} \ s2\text{-def}$
show $\text{snd } ?A ! \text{dfa-steps } ?A \ 0 \ bs = f (\text{snd } A ! \text{dfa-steps } A \ 0 \ bs) (\text{snd } B ! \text{dfa-steps } B \ 0 \ bs)$
by simp
qed
end

lemma binop-wf-dfa :
assumes $A: \text{wf-dfa } A \ n$ **and** $B: \text{wf-dfa } B \ n$
shows $\text{wf-dfa} (\text{binop-dfa } f \ A \ B) \ n$
proof –
from $A \ B$
interpret $\text{prod-DFS } A \ B \ n$ **by** unfold-locales
show $?thesis$ **by** $(\text{rule binop-wf-dfa})$
qed

theorem binop-dfa-accepts :
assumes $A: \text{wf-dfa } A \ n$
and $B: \text{wf-dfa } B \ n$

and X : *list-all* (*is-alph* n) bss
shows $dfa\text{-accepts}$ (*binop-dfa* f A B) $bss = f$ ($dfa\text{-accepts}$ A bss) ($dfa\text{-accepts}$ B bss)
proof –
from A B
interpret *prod-DFS* A B n **by** *unfold-locales*
from X **show** *?thesis*
by (*simp add: accepts-def dfa-accepting-def binop-dfa-steps*)
qed

definition

$and\text{-dfa} :: dfa \Rightarrow dfa \Rightarrow dfa$ **where**
 $and\text{-dfa} = \text{binop-dfa } (\wedge)$

lemma *and-wf-dfa*:

assumes *wf-dfa* M n
and *wf-dfa* N n
shows *wf-dfa* ($and\text{-dfa}$ M N) n
using *assms* **by** (*simp add: and-dfa-def binop-wf-dfa*)

lemma *and-dfa-accepts*:

assumes *wf-dfa* M n
and *wf-dfa* N n
and *list-all* (*is-alph* n) bs
shows $dfa\text{-accepts}$ ($and\text{-dfa}$ M N) $bs = (dfa\text{-accepts}$ M $bs \wedge dfa\text{-accepts}$ N $bs)$
using *assms* **by** (*simp add: binop-dfa-accepts and-dfa-def*)

definition

$or\text{-dfa} :: dfa \Rightarrow dfa \Rightarrow dfa$ **where**
 $or\text{-dfa} = \text{binop-dfa } (\vee)$

lemma *or-wf-dfa*:

assumes *wf-dfa* M n **and** *wf-dfa* N n
shows *wf-dfa* ($or\text{-dfa}$ M N) n
using *assms* **by** (*simp add: or-dfa-def binop-wf-dfa*)

lemma *or-dfa-accepts*:

assumes *wf-dfa* M n **and** *wf-dfa* N n
and *list-all* (*is-alph* n) bs
shows $dfa\text{-accepts}$ ($or\text{-dfa}$ M N) $bs = (dfa\text{-accepts}$ M $bs \vee dfa\text{-accepts}$ N $bs)$
using *assms* **by** (*simp add: binop-dfa-accepts or-dfa-def*)

definition

$imp\text{-dfa} :: dfa \Rightarrow dfa \Rightarrow dfa$ **where**
 $imp\text{-dfa} = \text{binop-dfa } (\longrightarrow)$

lemma *imp-wf-dfa*:

assumes *wf-dfa* M n **and** *wf-dfa* N n
shows *wf-dfa* ($imp\text{-dfa}$ M N) n

using *assms* by (*simp add: binop-wf-dfa imp-dfa-def*)

lemma *imp-dfa-accepts*:

assumes *wf-dfa M n* and *wf-dfa N n*

and *list-all (is-alph n) bs*

shows *dfa-accepts (imp-dfa M N) bs = (dfa-accepts M bs \longrightarrow dfa-accepts N bs)*

using *assms* by (*auto simp add: binop-dfa-accepts imp-dfa-def*)

5.3 Transforming DFAs to NFAs

definition

nfa-of-dfa :: *dfa* \Rightarrow *nfa* **where**

nfa-of-dfa = (λ (*bdd, as*). (*map (bdd-map* (λ q. (*replicate (length bdd) False*)[*q:=True*]))

bdd, as))

lemma *dfa2wf-nfa*:

assumes *wf-dfa M n*

shows *wf-nfa (nfa-of-dfa M) n*

proof –

have \bigwedge a. *dfa-is-node M a* \Longrightarrow *nfa-is-node (nfa-of-dfa M) ((replicate (length (fst M)) False)[a:=True])*

by (*simp add: dfa-is-node-def nfa-is-node-def nfa-of-dfa-def split-beta*)

hence \bigwedge *bdd*. *bdd-all (dfa-is-node M) bdd* \Longrightarrow *bdd-all (nfa-is-node (nfa-of-dfa M)) (bdd-map* (λ q. (*replicate (length (fst M)) False*)[*q:=True*]) *bdd*)

by (*simp add: bdd-all-bdd-map*)

with *assms* **have** *list-all (bdd-all (nfa-is-node (nfa-of-dfa M))) (fst (nfa-of-dfa M))* **by** (*simp add: list-all-iff split-beta nfa-of-dfa-def wf-dfa-def*)

with *assms* **show** *?thesis* **by** (*simp add: wf-nfa-def wf-dfa-def nfa-of-dfa-def split-beta list-all-iff bddh-bdd-map*)

qed

lemma *replicate-upd-inj*: $\llbracket q < n; (\text{replicate } n \text{ False})[q:=\text{True}] = (\text{replicate } n \text{ False})[p:=\text{True}] \Longrightarrow (q = p) \text{ (is } \llbracket - ; ?lhs = ?rhs \rrbracket \Longrightarrow -)$

proof –

assume *q: q < n* and *r: ?lhs = ?rhs*

{ assume *p \neq q*

with *q* **have** *?lhs ! q = True* **by** *simp*

moreover from $\langle p \neq q \rangle$ *q* **have** *?rhs ! q = False* **by** *simp*

ultimately have *?lhs \neq ?rhs* **by** *auto*

}

with *r* **show** *q = p* **by** *auto*

qed

lemma *nfa-of-dfa-reach'*:

assumes *V: wf-dfa M l*

and *X: list-all (is-alph l) bss*

and *N: n1 = (replicate (length (fst M)) False)[q:=True]*

and *Q: dfa-is-node M q*

and *R: nfa-reach (nfa-of-dfa M) n1 bss n2*

shows $\exists p. \text{dfa-reach } M \ q \ \text{bss } p \wedge n2 = (\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[p:=\text{True}]$
proof –
from $R \ V \ X \ N \ Q$ **show** *?thesis* **proof** *induct*
case Nil
hence $\text{dfa-reach } M \ q \ [] \ q$ **by** (*simp add: reach-nil*)
with Nil **show** *?case* **by** *auto*
next
case (*snoc j bss bs*)
hence $N1: \text{nfa-is-node } (\text{nfa-of-dfa } M) \ n1$ **by** (*simp add: nfa-is-node-def nfa-of-dfa-def split-beta*)
from *snoc* **have** $V2: \text{wf-nfa } (\text{nfa-of-dfa } M) \ l$ **by** (*simp add: dfa2wf-nfa*)
from *snoc* **have** $\exists p. \text{dfa-reach } M \ q \ \text{bss } p \wedge j = (\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[p := \text{True}]$ **by** *simp*
then obtain p **where** $PR: \text{dfa-reach } M \ q \ \text{bss } p$ **and** $J: j = (\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[p:=\text{True}]$ **by** *blast*
hence $JL: \text{nfa-is-node } (\text{nfa-of-dfa } M) \ j$ **by** (*simp add: nfa-is-node-def nfa-of-dfa-def split-beta*)
from *snoc PR* **have** $PL: \text{dfa-is-node } M \ p$ **by** (*simp add: dfa-reach-is-node*)
with *snoc JL* **have** $PL': p < \text{length } j$ **by** (*simp add: nfa-is-node-def dfa-is-node-def nfa-of-dfa-def split-beta*)
define m **where** $m = \text{dfa-trans } M \ p \ \text{bs}$
with *snoc PR* **have** $MR: \text{dfa-reach } M \ q \ (\text{bss } @ \ [\text{bs}]) \ m$ **by** (*simp add: reach-snoc*)
with *snoc* **have** $mL: \text{dfa-is-node } M \ m$ **by** (*simp add: dfa-reach-is-node*)
from $V2 \ JL \ \text{snoc}$ **have** $\text{nfa-is-node } (\text{nfa-of-dfa } M) \ (\text{nfa-trans } (\text{nfa-of-dfa } M) \ j \ \text{bs})$ **by** (*simp add: nfa-trans-is-node*)
hence $L: \text{length } (\text{nfa-trans } (\text{nfa-of-dfa } M) \ j \ \text{bs}) = \text{length } (\text{fst } M)$ **by** (*simp add: nfa-is-node-def nfa-of-dfa-def split-beta*)

have $\text{nfa-trans } (\text{nfa-of-dfa } M) \ j \ \text{bs} = (\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[m := \text{True}]$ (*is ?lhs = ?rhs*)
proof (*simp add: list-eq-iff-nth-eq L, intro strip*)
fix i **assume** $H: i < \text{length } (\text{fst } M)$
show $\text{nfa-trans } (\text{nfa-of-dfa } M) \ j \ \text{bs} ! i = (\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[m := \text{True}] ! i$ (*is ?lhs = ?rhs*)
proof
assume $\text{lhs}: ?\text{lhs}$
from $V2 \ \text{snoc}$ **have** $\text{wf-nfa } (\text{nfa-of-dfa } M) \ (\text{length } \text{bs})$ **by** (*simp add: is-alph-def*) **moreover**
note JL **moreover**
from H **have** $IL: i < \text{length } (\text{fst } (\text{nfa-of-dfa } M))$ **by** (*simp add: nfa-of-dfa-def split-beta*) **moreover**
from $\langle ?\text{lhs} \rangle$ **have** $\text{bdd-lookup } (\text{subsetbdd } (\text{fst } (\text{nfa-of-dfa } M)) \ j \ (\text{nfa-emptybdd } (\text{length } j))) \ \text{bs} ! i$ **by** (*simp add: nfa-trans-def*)
ultimately have $\exists x < \text{length } j. j ! x \wedge \text{bdd-lookup } (\text{fst } (\text{nfa-of-dfa } M)) ! x \ \text{bs} ! i$ **by** (*simp add: bdd-lookup-subsetbdd*)
then obtain x **where** $xl: x < \text{length } j$ **and** $xj: j ! x$ **and** $xs: \text{bdd-lookup } (\text{fst } (\text{nfa-of-dfa } M)) ! x \ \text{bs} ! i$ **by** *blast*
with *snoc J PL'* **have** $x = p$ **by** (*cases p = x simp+*)
with $xs \ PL \ \text{snoc}(3,4) \ m\text{-def}$ **show** $(\text{replicate } (\text{length } (\text{fst } M)) \ \text{False})[m :=$

```

True] ! i
  by (simp add: nfa-of-dfa-def split-beta dfa-trans-def dfa-is-node-def wf-dfa-def
is-alph-def bdd-map-bdd-lookup list-all-iff)
  next
    assume rhs: ?rhs
    with H mL have m = i by (cases m = i) (simp add: dfa-is-node-def)+
    from PL snoc(3,4) m-def ⟨m = i⟩ H have bdd-lookup (fst (nfa-of-dfa M) !
p) bs ! i
      by (simp add: nfa-of-dfa-def split-beta dfa-is-node-def wf-dfa-def is-alph-def
list-all-iff bdd-map-bdd-lookup dfa-trans-def)
      with PL' J have E: ∃ p < length j. j ! p ∧ bdd-lookup (fst (nfa-of-dfa M) !
p) bs ! i by auto
    from snoc(4) V2 have V': wf-nfa (nfa-of-dfa M) (length bs) by (simp add:
is-alph-def)
    from H have H': i < length (fst (nfa-of-dfa M)) by (simp add: nfa-of-dfa-def
split-beta)
    from H' V' E JL have bdd-lookup (subsetbdd (fst (nfa-of-dfa M)) j
(nfa-emptybdd (length j))) bs ! i by (simp add: bdd-lookup-subsetbdd)
    thus ?lhs by (simp add: nfa-trans-def)
  qed
  qed
  with MR show ?case by auto
  qed
  qed

```

lemma *nfa-of-dfa-reach*:

```

assumes V: wf-dfa M l
and X: list-all (is-alph l) bss
and N1: n1 = (replicate (length (fst M)) False)[q:=True]
and N2: n2 = (replicate (length (fst M)) False)[p:=True]
and Q: dfa-is-node M q
shows nfa-reach (nfa-of-dfa M) n1 bss n2 = dfa-reach M q bss p
proof
  assume nfa-reach (nfa-of-dfa M) n1 bss n2
  with assms have ∃ p. dfa-reach M q bss p ∧ n2 = (replicate (length (fst M))
False)[p := True] by (simp add: nfa-of-dfa-reach')
  then obtain p' where R: dfa-reach M q bss p' and N2': n2 = (replicate (length
(fst M)) False)[p' := True] by blast
  from V R Q X have dfa-is-node M p' by (simp add: dfa-reach-is-node)
  with N2 N2' have p' = p by (simp add: dfa-is-node-def replicate-upd-inj)
  with R show dfa-reach M q bss p by simp
next
  assume H: dfa-reach M q bss p
  define n2' where n2' = nfa-steps (nfa-of-dfa M) n1 bss
  hence R': nfa-reach (nfa-of-dfa M) n1 bss n2' by (simp add: reach-def)
  with assms have ∃ p. dfa-reach M q bss p ∧ n2' = (replicate (length (fst M))
False)[p := True] by (simp add: nfa-of-dfa-reach')
  then obtain p' where R: dfa-reach M q bss p' and N2': n2' = (replicate (length

```

(fst M)) False][p' := True] **by** blast
with H **have** p = p' **by** (simp add: reach-inj)
with N2' N2 **have** n2 = n2' **by** simp
with R' **show** nfa-reach (nfa-of-dfa M) n1 bss n2 **by** simp
qed

lemma nfa-accepting-replicate:
assumes q < length (fst N)
and length (snd N) = length (fst N)
shows nfa-accepting N ((replicate (length (fst N)) False)[q:=True]) = snd N ! q
proof –
from assms **have** set-of-bv ((replicate (length (fst N)) False)[q:=True]) = {q}
proof (auto simp: set-of-bv-def)
fix x **assume** x < length (fst N) **and** (replicate (length (fst N)) False)[q := True] ! x
with assms **show** x = q **by** (cases x = q) simp+
qed
hence nfa-accepting N ((replicate (length (fst N)) False)[q:=True]) = (set-of-bv (snd N) ∩ {q} ≠ {})
by (simp add: nfa-accepting-set-of-bv)
also **have** ... = (q ∈ set-of-bv (snd N)) **by** auto
also **from** assms **have** ... = snd N ! q **by** (auto simp: set-of-bv-def)
finally **show** ?thesis .
qed

lemma nfa-of-dfa-accepts:
assumes V: wf-dfa A n
and X: list-all (is-alph n) bss
shows nfa-accepts (nfa-of-dfa A) bss = dfa-accepts A bss
proof –
from V **have** Q: dfa-is-node A 0 **by** (simp add: dfa-startnode-is-node)
have S: nfa-startnode (nfa-of-dfa A) = (replicate (length (fst A)) False)[0:= True]
by (simp add: nfa-startnode-def nfa-of-dfa-def split-beta)
define p **where** p = dfa-steps A 0 bss
define n2 **where** n2 = (replicate (length (fst A)) False)[p := True]
from p-def **have** PR: dfa-reach A 0 bss p **by** (simp add: reach-def)
with p-def n2-def Q S X V **have** nfa-reach (nfa-of-dfa A) (nfa-startnode (nfa-of-dfa A)) bss n2 **by** (simp add: nfa-of-dfa-reach)
hence N2: n2 = nfa-steps (nfa-of-dfa A) (nfa-startnode (nfa-of-dfa A)) bss **by** (simp add: reach-def)
from PR Q X V **have** dfa-is-node A p **by** (simp add: dfa-reach-is-node)
hence p < length (fst (nfa-of-dfa A)) **by** (simp add: dfa-is-node-def nfa-of-dfa-def split-beta) **moreover**
from dfa2wf-nfa[OF V] **have** length (snd (nfa-of-dfa A)) = length (fst (nfa-of-dfa A)) **by** (auto simp add: wf-nfa-def) **moreover**
from n2-def **have** n2 = (replicate (length (fst (nfa-of-dfa A))) False)[p := True]
by (simp add: nfa-of-dfa-def split-beta)
ultimately **have** nfa-accepting (nfa-of-dfa A) n2 = snd (nfa-of-dfa A) ! p **by** (simp add: nfa-accepting-replicate)

with $N2$ p -def **show** $?thesis$ **by** (*simp add: accepts-def accepts-def dfa-accepting-def nfa-of-dfa-def split-beta*)

qed

5.4 Transforming NFAs to DFAs

fun

bddinsert :: 'a bdd \Rightarrow bool list \Rightarrow 'a \Rightarrow 'a bdd

where

bddinsert (Leaf a) [] $x =$ Leaf x
| *bddinsert* (Leaf a) (w#ws) $x =$ (if w then Branch (Leaf a) (*bddinsert* (Leaf a) ws x) else Branch (*bddinsert* (Leaf a) ws x) (Leaf a))
| *bddinsert* (Branch l r) (w#ws) $x =$ (if w then Branch l (*bddinsert* r ws x) else Branch (*bddinsert* l ws x) r)

lemma *bddh-bddinsert*:

assumes *bddh* x b

and *length* $w \geq x$

shows *bddh* (*length* w) (*bddinsert* b w y)

using *assms* **proof** (*induct* b w y *arbitrary*; x *rule: bddinsert.induct*)

case (2 aa ww wss yy xaa)

have *bddh* 0 (Leaf aa) \wedge $0 \leq$ *length* wss **by** *simp*

with 2(1) 2(2) **have** *bddh* (*length* wss) (*bddinsert* (Leaf aa) wss yy) **by** (*cases* ww) *blast+*

with 2 **show** $?case$ **by** *simp*

next

case (3 ll rr ww wss yy xx)

from 3(3) **obtain** y **where** $Y: Suc$ $y = xx$ **by** (*cases* xx) *simp+*

with 3 **have** 1: *bddh* y $rr \wedge$ *bddh* y $ll \wedge$ $y \leq$ *length* wss **by** *auto*

show $?case$ **proof** (*cases* ww)

case True

with 1 3(1) **have** IV: *bddh* (*length* wss) (*bddinsert* rr wss yy) **by** *blast*

with Y 3 **have** $y \leq$ *length* wss **and** *bddh* y ll **by** *auto*

hence *bddh* (*length* wss) ll **by** (*rule* *bddh-ge*)

with IV True **show** $?thesis$ **by** *simp*

next

case False

with 1 3(2) **have** IV: *bddh* (*length* wss) (*bddinsert* ll wss yy) **by** *blast*

with Y 3 **have** $y \leq$ *length* wss **and** *bddh* y rr **by** *auto*

hence *bddh* (*length* wss) rr **by** (*rule* *bddh-ge*)

with IV False **show** $?thesis$ **by** *simp*

qed

qed *simp+*

lemma *bdd-lookup-bddinsert*:

assumes *bddh* (*length* w) bd

and *length* $w =$ *length* v

shows *bdd-lookup* (*bddinsert* bd w y) $v =$ (if $w = v$ then y else *bdd-lookup* bd v)

using *assms* **proof** (*induct* bd w y *arbitrary*; v *rule: bddinsert.induct*)

case (2 aa ww wss xx vv)
hence $\exists v \text{ vs. } vv = v \# \text{ vs}$ **by** (cases vv) simp+
then obtain v vs **where** V: vv = v # vs **by** blast
with 2 have length wss = length vs **by** simp
with 2 have IV: bdd-lookup (bddinsert (Leaf aa) wss xx) vs = (if wss = vs then xx else bdd-lookup (Leaf aa) vs) **by** (cases ww) simp+
have bdd-lookup (bddinsert (Leaf aa) (ww # wss) xx) vv = bdd-lookup (if ww then (Branch (Leaf aa) (bddinsert (Leaf aa) wss xx)) else Branch (bddinsert (Leaf aa) wss xx) (Leaf aa)) vv **by** simp
also have ... = (if ww then bdd-lookup (Branch (Leaf aa) (bddinsert (Leaf aa) wss xx)) vv else bdd-lookup (Branch (bddinsert (Leaf aa) wss xx) (Leaf aa)) vv) **by** simp
also from V IV **have** ... = (if ww # wss = v # vs then bdd-lookup (bddinsert (Leaf aa) wss xx) vs else bdd-lookup (Leaf aa) vs) **by** (cases ww) auto
also from V IV **have** ... = (if ww # wss = vv then xx else bdd-lookup (Leaf aa) vs) **by** auto
finally show ?case **by** simp
next
case (3 ll rr ww wss xx vv)
hence $\exists v \text{ vs. } vv = v \# \text{ vs}$ **by** (cases vv) simp+
then obtain v vs **where** V: vv = v # vs **by** blast
show ?case **proof** (cases ww)
case True
with 3 V have IV: bdd-lookup (bddinsert rr wss xx) vs = (if wss = vs then xx else bdd-lookup rr vs) **by** simp
with True 3 V **show** ?thesis **by** auto
next
case False
with 3 V have IV: bdd-lookup (bddinsert ll wss xx) vs = (if wss = vs then xx else bdd-lookup ll vs) **by** simp
with False 3 V **show** ?thesis **by** auto
qed
qed simp+

definition

subset-succs :: nfa \Rightarrow bool list \Rightarrow bool list list **where**
subset-succs A qs = add-leaves (subsetbdd (fst A) qs (nfa-emptybdd (length qs)))
□

definition

subset-invariant :: nfa \Rightarrow nat option bdd \times bool list list \Rightarrow bool **where**
subset-invariant A = (λ (bdd, qss). bddh (length (fst A)) bdd)

definition

subset-ins qs = (λ (bdd, qss). (bddinsert bdd qs (Some (length qss)), qss @ [qs]))

definition

subset-memb :: bool list \Rightarrow nat option bdd \times bool list list \Rightarrow bool **where**
subset-memb qs = (λ (bdd, qss). bdd-lookup bdd qs \neq None)

definition

subset-empt :: nat option bdd × bool list list **where**
subset-empt = (Leaf None, [])

definition

subset-dfs :: nfa ⇒ bool list ⇒ nat option bdd × bool list list **where**
subset-dfs A x = gen-dfs (subset-succs A) subset-ins subset-memb subset-empt [x]

definition

det-nfa :: nfa ⇒ dfa **where**
det-nfa A = (let (bdd, qss) = subset-dfs A (nfa-startnode A) in
 (map (λqs. bdd-map (λqs. the (bdd-lookup bdd qs)) (subsetbdd (fst A) qs
 (nfa-emptybdd (length qs)))) qss,
 map (λqs. nfa-accepting A qs) qss))

locale *subset-DFS* =

fixes A n
assumes *well-formed*: wf-nfa A n

lemma *finite-list*: finite {xs::('a::finite) list. length xs = k}

apply (induct k)
apply simp
apply (subgoal-tac {xs::('a::finite) list. length xs = Suc k} = (∪ x. Cons x ‘ {xs.
length xs = k}))
apply auto
apply (case-tac x)
apply auto
done

sublocale *subset-DFS* < DFS subset-succs A nfa-is-node A subset-invariant A
subset-ins subset-memb subset-empt

apply (unfold-locales)
apply (simp add: nfa-is-node-def subset-invariant-def subset-memb-def subset-ins-def
bdd-lookup-bddinsert split-beta)
apply (simp add: nfa-is-node-def subset-memb-def subset-empt-def)

apply (insert well-formed)[]
apply (simp add: subset-succs-def add-leaves-bdd-all-eq bdd-all-is-node-subsetbdd
wf-nfa-def)

apply (simp add: subset-invariant-def subset-empt-def)

apply (simp add: nfa-is-node-def subset-invariant-def subset-memb-def subset-ins-def
split-paired-all)

apply (subgoal-tac length (fst A) = length x)

apply (auto simp: bddh-bddinsert)

apply (simp add: nfa-is-node-def)

```

apply (rule finite-list)
done

context subset-DFS
begin

lemmas dfs-eq-rtrancl[folded subset-dfs-def] = dfs-eq-rtrancl

lemma subset-dfs-bij:
  assumes H1: nfa-is-node A q
  and H2: nfa-is-node A q0
  shows (bdd-lookup (fst (subset-dfs A q0)) q = Some v) = (v < length (snd
(subset-dfs A q0)) ∧ (snd (subset-dfs A q0)) ! v = q)
proof –
  from assms have list-all (nfa-is-node A) [q0] by simp
  with empt-invariant show ?thesis using H1 unfolding subset-dfs-def
  proof (induct arbitrary: v q rule: dfs-invariant)
    case (step S x vv qq)
    obtain bd1 l1 where S: S = (bd1, l1) by (cases S) blast+
    { assume x ∈ set l1
      hence list-ex (λl. l = x) l1 by (simp add: list-ex-iff)
      hence ∃ i < length l1. l1 ! i = x by (simp add: list-ex-length)
      then obtain i where i < length l1 ∧ l1 ! i = x by blast
      with step S have bdd-lookup bd1 x = Some i by (simp add: nfa-is-node-def)
      with step S have False by (simp add: subset-memb-def) }
    hence X: ∀ i < length l1. l1 ! i ≠ x by auto
    obtain bd2 l2 where S2: subset-ins x S = (bd2, l2) by (cases subset-ins x S)
blast+
    with S have SS: bd2 = bddinsert bd1 x (Some (length l1)) l2 = l1 @ [x] by
(simp add: subset-ins-def)+
    from step S H1 have bdd-lookup (bddinsert bd1 x (Some (length l1))) qq = (if
x = qq then Some (length l1) else bdd-lookup bd1 qq)
    by (simp add: bdd-lookup-bddinsert subset-invariant-def nfa-is-node-def)
    with SS have (bdd-lookup bd2 qq = Some vv) = (if x = qq then length l1 = vv
else bdd-lookup bd1 qq = Some vv) by simp
    also have ... = (x = qq ∧ length l1 = vv ∨ x ≠ qq ∧ bdd-lookup bd1 qq =
Some vv) by auto
    also have ... = (vv < length l2 ∧ l2 ! vv = qq) proof (cases x = qq)
      case True
      hence (x = qq ∧ length l1 = vv ∨ x ≠ qq ∧ bdd-lookup bd1 qq = Some vv)
= (x = qq ∧ length l1 = vv) by simp
      also have ... = (vv < length l2 ∧ l2 ! vv = qq) proof
        assume H: vv < length l2 ∧ l2 ! vv = qq
        show x = qq ∧ length l1 = vv proof (cases vv = length l1)
          case False
          with H SS have vv < length l1 by simp
          with SS have l2 ! vv = l1 ! vv by (simp add: nth-append)
          with False H SS ⟨x = qq⟩ have vv < length l1 ∧ l1 ! vv = x by auto
          with X show ?thesis by auto
        case True
        with H SS have vv = length l1 by simp
        with SS have l2 ! vv = l1 ! vv by (simp add: nth-append)
        with True H SS ⟨x = qq⟩ have vv < length l1 ∧ l1 ! vv = x by auto
        with X show ?thesis by auto
      case False
      with H SS have vv < length l2 ∧ l2 ! vv = qq by simp
      with SS have l2 ! vv = l1 ! vv by (simp add: nth-append)
      with False H SS ⟨x = qq⟩ have vv < length l1 ∧ l1 ! vv = x by auto
      with X show ?thesis by auto
    }
  qed

```



```

      qed (simp add: True)
    qed (auto simp: SS)
  finally show ?thesis .
next
  case False
  hence (x = qq ∧ length l1 = vv ∨ x ≠ qq ∧ bdd-lookup bd1 qq = Some vv)
= (x ≠ qq ∧ bdd-lookup bd1 qq = Some vv) by simp
  also from step(4,5) S ⟨x≠qq⟩ have ... = (vv < length l1 ∧ l1 ! vv = qq) by
simp
  also from SS ⟨x≠qq⟩ have ... = (vv < length l2 ∧ l2 ! vv = qq) by (simp
add: nth-append)
  finally show ?thesis .
qed
  finally show ?case by (simp add: S2)
qed (simp add: subset-empt-def)
qed

```

lemma subset-dfs-start:

```

  assumes H: nfa-is-node A q0
  shows bdd-lookup (fst (subset-dfs A q0)) q0 = Some 0
proof -
  obtain bd l where S: subset-ins q0 subset-empt = (bd, l) by (cases subset-ins
q0 subset-empt) blast+
  from H have ¬ subset-memb q0 subset-empt by (simp add: empt)
  with H empt-invariant have I: subset-invariant A (subset-ins q0 subset-empt)
  by (simp add: ins-invariant)
  from H have list-all (nfa-is-node A) (subset-succs A q0) by (simp add: succs-is-node)
  with I have bdd-lookup (fst (gen-dfs (subset-succs A) subset-ins subset-memb
(subset-ins q0 subset-empt) (subset-succs A q0))) q0 = Some 0
  proof (induct rule: dfs-invariant)
    case base thus ?case unfolding subset-ins-def subset-empt-def by (induct q0)
simp+
  next
    case (step S x)
    hence Q: subset-memb q0 S by (simp add: subset-memb-def split-beta)
    with step have q0 ≠ x by auto
    from step have I: bddh (length (fst A)) (fst S) by (simp add: subset-invariant-def
split-beta)
    with H step ⟨q0≠x⟩ have V: ⋀v. bdd-lookup (bddinsert (fst S) x v) q0 =
bdd-lookup (fst S) q0 by (simp add: bdd-lookup-bddinsert nfa-is-node-def)
    with step show bdd-lookup (fst (subset-ins x S)) q0 = Some 0 by (auto simp:
subset-ins-def split-beta)
  qed
  thus ?thesis unfolding subset-dfs-def by (auto simp: nfa-is-node-def gen-dfs-simps
subset-memb-def subset-empt-def)
qed

```

lemma subset-dfs-is-node:

```

  assumes nfa-is-node A q0

```

shows $list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ (snd\ (subset\text{-}dfs\ A\ q0))$
proof –
from $assms$ **have** $list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ [q0]$ **by** $simp$
with $empt\text{-}invariant$ **show** $?thesis$ **unfolding** $subset\text{-}dfs\text{-}def$
proof ($induct\ rule: dfs\text{-}invariant$)
case $base$ **thus** $?case$ **by** ($simp\ add: subset\text{-}empt\text{-}def$)
next
case ($step\ S\ x$) **thus** $?case$ **by** ($simp\ add: subset\text{-}ins\text{-}def\ split\text{-}beta$)
qed
qed

lemma $det\text{-}wf\text{-}nfa$:

shows $wf\text{-}dfa\ (det\text{-}nfa\ A)\ n$
proof –
obtain $bt\ ls$ **where** $BT: subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A) = (bt, ls)$ **by** ($cases\ subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A)$) $auto$
note $Q = nfa\text{-}startnode\text{-}is\text{-}node[OF\ well\text{-}formed]$
from Q **have** $N: list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ (snd\ (subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A)))$
by ($simp\ add: subset\text{-}dfs\text{-}is\text{-}node$)
with BT **have** $L: list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ ls$ **by** $simp$
have $D: det\text{-}nfa\ A = (map\ (\lambda q. bdd\text{-}map\ (\lambda q. the\ (bdd\text{-}lookup\ bt\ q))\ (subsetbdd\ (fst\ A)\ q\ (nfa\text{-}emptybdd\ (length\ q))))\ ls, map\ (\lambda q. nfa\text{-}accepting\ A\ q)\ ls)$
(is $- = (?bdt, ?atbl)$ **unfolding** $det\text{-}nfa\text{-}def$ **by** ($simp\ add: BT$)
from $well\text{-}formed\ L$ **have** $list\text{-}all\ (\lambda q. bddh\ n\ (subsetbdd\ (fst\ A)\ q\ (nfa\text{-}emptybdd\ (length\ q))))\ ls$
by ($induct\ ls$) ($simp\ add: bddh\text{-}subsetbdd\ wf\text{-}nfa\text{-}def\ nfa\text{-}emptybdd\text{-}def$)
hence $list\text{-}all\ (\lambda q. bddh\ n\ (bdd\text{-}map\ (\lambda q. the\ (bdd\text{-}lookup\ bt\ q))\ (subsetbdd\ (fst\ A)\ q\ (nfa\text{-}emptybdd\ (length\ q))))\ ls$
by ($simp\ add: bddh\text{-}bdd\text{-}map$)
hence $A: list\text{-}all\ (bddh\ n)\ ?bdt$ **by** ($simp\ add: list\text{-}all\text{-}iff$)
{
fix q **assume** $\exists i < length\ ls. ls\ !\ i = q$
then obtain i **where** $len\text{-}i: i < length\ ls$ **and** $i: q = ls\ !\ i$ **by** $blast$
from $len\text{-}i\ i\ L$ **have** $Q': nfa\text{-}is\text{-}node\ A\ q$ **by** ($simp\ add: list\text{-}all\text{-}iff$)
then have $(bdd\text{-}lookup\ (fst\ (subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A)))\ q = Some\ i) =$
 $(i < length\ (snd\ (subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A))) \wedge snd\ (subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A))\ !\ i = q)$
using Q
by ($rule\ subset\text{-}dfs\text{-}bij$)
with $BT\ len\text{-}i\ i$ **have** $bdd\text{-}lookup\ bt\ q = Some\ i$ **by** $simp$
with BT **have** $subset\text{-}memb\ q\ (subset\text{-}dfs\ A\ (nfa\text{-}startnode\ A))$ **by** ($simp\ add: subset\text{-}memb\text{-}def$)
with $Q'\ Q$ **have** $TR: (nfa\text{-}startnode\ A, q) \in (succsr\ (subset\text{-}succs\ A))^*$
by ($simp\ add: dfs\text{-}eq\text{-}rtrancl$)
{
fix p **assume** $P: p \in set\ (subset\text{-}succs\ A\ q)$
with TR **have** $3: (nfa\text{-}startnode\ A, p) \in (succsr\ (subset\text{-}succs\ A))^*$ **by** ($simp\ add: succsr\text{-}def\ rtrancl\text{-}into\text{-}rtrancl$)
from Q' **have** $list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ (subset\text{-}succs\ A\ q)$

by (*rule succs-is-node*)
with P **have** 4: *nfa-is-node* A p **by** (*simp add: list-all-iff*)
with Q 3 **have** *subset-memb* p (*subset-dfs* A (*nfa-startnode* A))
by (*simp add: dfs-eq-rtrancl*)
with BT **have** *bdd-lookup* bt $p \neq \text{None}$ **by** (*simp add: subset-memb-def*)
with BT **obtain** j **where** j : *bdd-lookup* (*fst* (*subset-dfs* A (*nfa-startnode* A)))
 $p = \text{Some } j$ **by** (*cases bdd-lookup bt p*) *simp+*
from 4 Q j **have** $j < \text{length}$ (*snd* (*subset-dfs* A (*nfa-startnode* A))) \wedge (*snd* (*subset-dfs* A (*nfa-startnode* A))) ! $j = p$
by (*auto simp add: subset-dfs-bij*)
with j BT 4 **have** $\exists j. \text{bdd-lookup } bt \ p = \text{Some } j \wedge j < \text{length } ls$ **by** *auto*
}
hence $\forall p \in \text{set}$ (*subset-succs* A q). $\exists j. \text{bdd-lookup } bt \ p = \text{Some } j \wedge j < \text{length } ls$ **by** *auto*
hence *list-all* ($\lambda p. \exists j. \text{bdd-lookup } bt \ p = \text{Some } j \wedge j < \text{length } ls$) (*add-leaves* (*subsetbdd* (*fst* A) q (*nfa-emptybdd* (*length* q)))) [] **by** (*simp add: list-all-iff subset-succs-def*)
hence *bdd-all* ($\lambda p. \exists j. \text{bdd-lookup } bt \ p = \text{Some } j \wedge j < \text{length } ls$) (*subsetbdd* (*fst* A) q (*nfa-emptybdd* (*length* q)))) **by** (*simp add: add-leaves-bdd-all-eq*)
hence *bdd-all* ($\lambda l. l < \text{length } ls$) (*bdd-map* ($\lambda q. \text{the} (\text{bdd-lookup } bt \ q)$) (*subsetbdd* (*fst* A) q (*nfa-emptybdd* (*length* q))))
by (*induct* (*subsetbdd* (*fst* A) q (*nfa-emptybdd* (*length* q)))) *auto*
}
then **have** $\forall x \in \text{set } ls. \text{bdd-all}$ ($\lambda l. l < \text{length } ls$) (*bdd-map* ($\lambda q. \text{the} (\text{bdd-lookup } bt \ q)$) (*subsetbdd* (*fst* A) x (*nfa-emptybdd* (*length* x)))) **by** (*simp add: in-set-conv-nth*)
hence *list-all* ($\lambda x. \text{bdd-all}$ ($\lambda l. l < \text{length } ls$) (*bdd-map* ($\lambda q. \text{the} (\text{bdd-lookup } bt \ q)$) (*subsetbdd* (*fst* A) x (*nfa-emptybdd* (*length* x)))))) ls **by** (*simp add: list-all-iff*)
hence B : *list-all* (*bdd-all* ($\lambda l. l < \text{length } ls$) (*map* ($\lambda x. \text{bdd-map}$ ($\lambda q. \text{the} (\text{bdd-lookup } bt \ q)$) (*subsetbdd* (*fst* A) x (*nfa-emptybdd* (*length* x)))))) ls) **by** (*simp add: list-all-iff*)
from *well-formed* **have** *bdd-lookup* (*fst* (*subset-dfs* A (*nfa-startnode* A))) (*nfa-startnode* A) = *Some* 0
by (*simp add: subset-dfs-start nfa-startnode-is-node*)
with *well-formed* **have** $0 < \text{length}$ (*snd* (*subset-dfs* A (*nfa-startnode* A)))
by (*simp add: subset-dfs-bij nfa-startnode-is-node*)
with A B D BT **show** ?thesis **by** (*simp add: wf-dfa-def det-nfa-def dfa-is-node-def*)
qed

lemma *nfa-reach-rtrancl*:

assumes *nfa-is-node* A i

shows ($\exists \text{bss}. \text{nfa-reach } A \ i \ \text{bss } j \wedge \text{list-all} (\text{is-alph } n) \ \text{bss}$) = ($(i, j) \in (\text{succsr} (\text{subset-succs } A))^*$)

proof

assume $\exists \text{bss}. \text{nfa-reach } A \ i \ \text{bss } j \wedge \text{list-all} (\text{is-alph } n) \ \text{bss}$

then **obtain** bss **where** BS : *nfa-reach* $A \ i \ \text{bss } j$ *list-all* (*is-alph* n) bss **by** *blast*

show $(i, j) \in (\text{succsr} (\text{subset-succs } A))^*$

proof –

from BS **show** $(i, j) \in (\text{succsr} (\text{subset-succs } A))^*$ **proof** *induct*

case (*snoc* j bss bs)

with *assms well-formed* **have** J : *nfa-is-node* $A \ j$ **by** (*simp add: nfa-reach-is-node*)

with *snoc well-formed* **have** *bddh n (subsetbdd (fst A) j (nfa-emptybdd (length j)))*
by (*simp add: wf-nfa-def bddh-subsetbdd nfa-emptybdd-def*)
with *snoc(3)* **have** *bdd-lookup (subsetbdd (fst A) j (nfa-emptybdd (length j)))*
bs ∈ set (add-leaves (subsetbdd (fst A) j (nfa-emptybdd (length j))) [])
by (*auto simp: add-leaves-bdd-lookup*)
hence (*j, bdd-lookup (subsetbdd (fst A) j (nfa-emptybdd (length j))) bs*) ∈
(*succsr (subset-succs A)*)*
by (*auto simp: succsr-def subset-succs-def*)
with *snoc* **show** *?case* **by** (*simp add: nfa-trans-def*)
qed *simp*
qed
next
assume *ij: (i,j) ∈ (succsr (subset-succs A))**
from *ij* **show** \exists *bss. nfa-reach A i bss j \wedge *list-all (is-alph n) bss*
proof *induct*
case *base*
from *reach-nil[of nfa-trans A i]* **show** *?case* **by** *auto*
next
case (*step y z*)
then obtain *bss* **where** *BS: nfa-reach A i bss y* *list-all (is-alph n) bss* **by** *blast*
from *assms well-formed BS* **have** *nfa-is-node A y* **by** (*simp add: nfa-reach-is-node*)
with *well-formed BS* **have** *B: bddh n (subsetbdd (fst A) y (nfa-emptybdd (length y)))*
by (*simp add: wf-nfa-def bddh-subsetbdd nfa-emptybdd-def*)
from *step* **have** *z ∈ set (add-leaves (subsetbdd (fst A) y (nfa-emptybdd (length y))) [])* **by** (*simp add: succsr-def subset-succs-def*)
with *B* **have** \exists *bs. z = bdd-lookup (subsetbdd (fst A) y (nfa-emptybdd (length y))) bs \wedge *is-alph n bs* **by** (*simp add: add-leaves-bdd-lookup*)
then obtain *bs* **where** *Z: z = bdd-lookup (subsetbdd (fst A) y (nfa-emptybdd (length y))) bs* **and** *L: is-alph n bs* **by** *blast*
from *BS(1) L* **have** *nfa-reach A i (bss @ [bs]) (nfa-trans A y bs)* **by** (*simp add: reach-snoc*)
with *Z* **have** *nfa-reach A i (bss @ [bs]) z* **by** (*simp add: nfa-trans-def*) **more-over**
from *BS L* **have** *list-all (is-alph n) (bss @ [bs])* **by** *simp*
moreover **note** *BS(2) L*
ultimately **show** *?case* **by** *auto*
qed
qed**

lemma *nfa-reach-subset-memb:*

assumes *R: nfa-reach A q0 bss q*
and *Q0: nfa-is-node A q0*
and *X: list-all (is-alph n) bss*
shows *subset-memb q (subset-dfs A q0)*

proof –

from *assms well-formed* **have** *Q: nfa-is-node A q* **by** (*simp add: nfa-reach-is-node*)
from *R X* **have** \exists *bs. nfa-reach A q0 bs q \wedge *list-all (is-alph n) bs* **by** *auto**

with $Q0$ have $(q0, q) \in (\text{succsr } (\text{subset-succs } A))^*$ by (simp add: nfa-reach-rtrancl)
 with $Q0$ Q show ?thesis by (simp add: dfs-eq-rtrancl)
 qed

lemma *det-nfa-reach'*:

fixes $bd :: \text{nat option bdd}$ and $ls :: \text{bool list list}$
 assumes $\text{subset-dfs } A (\text{nfa-startnode } A) = (bd, ls)$ (is ?subset-dfs = -)
 and $\exists bs. \text{nfa-reach } A (\text{nfa-startnode } A) bs \ q1 \wedge \text{list-all } (\text{is-alph } n) bs$
 and $q1 = ls ! i$ and $q2 = ls ! j$ and $i < \text{length } ls$ and $j < \text{length } ls$
 and $\text{list-all } (\text{is-alph } n) bss$
 shows $\text{nfa-reach } A \ q1 \ bss \ q2 = (\text{dfa-reach } (\text{det-nfa } A) \ i \ bss \ j \wedge \text{nfa-is-node } A \ q2)$
 (is - = (dfa-reach ?M i bss j \wedge -))
 proof
 assume $\text{nfa-reach } A \ q1 \ bss \ q2$
 from this assms show $\text{dfa-reach } ?M \ i \ bss \ j \wedge \text{nfa-is-node } A \ q2$
 proof (induct arbitrary: j)
 case (Nil j)
 with well-formed have $Q0: \text{nfa-is-node } A (\text{nfa-startnode } A)$ by (simp add: nfa-startnode-is-node)
 from Nil obtain bs where $\text{nfa-reach } A (\text{nfa-startnode } A) bs \ q1$ and $\text{list-all } (\text{is-alph } n) bs$ by blast
 with well-formed $Q0$ Nil have $Q1: \text{nfa-is-node } A \ q1$ by (simp add: nfa-reach-is-node)
 with $Q0$ have $\bigwedge v. (\text{bdd-lookup } (\text{fst } (?subset-dfs)) \ q1 = \text{Some } v) = (v < \text{length } (\text{snd } (?subset-dfs)) \wedge \text{snd } (?subset-dfs) ! v = q1)$
 by (simp add: subset-dfs-bij)
 with Nil(1) have 1: $\bigwedge v. (\text{bdd-lookup } bd \ q1 = \text{Some } v) = (v < \text{length } ls \wedge ls ! v = q1)$ by simp
 from Nil 1 have $\text{bdd-lookup } bd \ q1 = \text{Some } i$ by simp
 moreover from Nil 1 have $\text{bdd-lookup } bd \ q1 = \text{Some } j$ by simp
 ultimately have $i = j$ by simp
 have $\text{dfa-reach } ?M \ i \ [] \ i$ by (simp add: reach-nil)
 with $\langle i=j \rangle \ Q1$ show ?case by simp
 next
 case (snoc p bss bs j)
 note $S\text{-len} = \text{nfa-startnode-is-node}[OF \text{well-formed}]$
 from snoc obtain bss' where $BSS': \text{nfa-reach } A (\text{nfa-startnode } A) \ bss' \ q1$ and $BSS'L: \text{list-all } (\text{is-alph } n) \ bss'$ by blast
 with well-formed $S\text{-len}$ have $Q\text{-len}: \text{nfa-is-node } A \ q1$ by (simp add: nfa-reach-is-node)
 with well-formed snoc have $P\text{-len}: \text{nfa-is-node } A \ p$ by (simp add: nfa-reach-is-node)
 from BSS' snoc have $\text{nfa-reach } A (\text{nfa-startnode } A) (bss' @ bss) \ p$ by (simp add: reach-trans) moreover
 note $S\text{-len}$ moreover
 from snoc $BSS'L$ have $\text{list-all } (\text{is-alph } n) (bss' @ bss)$ by simp
 ultimately have $\text{subset-memb } p \ ?subset-dfs$ by (rule nfa-reach-subset-memb)
 hence $\text{bdd-lookup } (\text{fst } ?subset-dfs) \ p \neq \text{None}$ by (simp add: subset-memb-def split-beta)
 then obtain v where $P: \text{bdd-lookup } (\text{fst } ?subset-dfs) \ p = \text{Some } v$ by (cases $\text{bdd-lookup } (\text{fst } ?subset-dfs) \ p$) simp+
 with $P\text{-len } S\text{-len}$ have $v < \text{length } (\text{snd } (?subset-dfs)) \wedge \text{snd } ?subset-dfs ! v =$

p by (simp add: subset-dfs-bij)
with snoc have $V: v < \text{length } ls \wedge ls ! v = p$ **by** simp
with snoc P-len have $R: \text{dfa-reach } ?M \ i \ \text{bss } v \wedge \text{nfa-is-node } A \ p$ **by** simp

from snoc have $BS: \text{is-alph } n \ \text{bs}$ **by** simp
with well-formed P-len have $Z: \text{nfa-is-node } A \ (\text{nfa-trans } A \ p \ \text{bs})$ **by** (simp add: nfa-trans-is-node)

with snoc have $N: \text{nfa-is-node } A \ (ls ! j)$ **by** simp
from snoc have $j < \text{length } (snd \ ?subset-dfs) \wedge \text{snd } ?subset-dfs ! j = ls ! j$ **by** simp
with N S-len have $\text{bdd-lookup } (fst \ ?subset-dfs) \ (ls ! j) = \text{Some } j$ **by** (simp add: subset-dfs-bij)
with snoc have $J: \text{bdd-lookup } bd \ (ls ! j) = \text{Some } j$ **by** simp

from snoc have $BD: \text{fst } ?M = \text{map } (\lambda q. \text{bdd-map } (\lambda q. \text{the } (\text{bdd-lookup } bd \ q)) \ (\text{subsetbdd } (fst \ A) \ q \ (\text{nfa-emptybdd } (\text{length } q)))) \ ls$
by (simp add: det-nfa-def)
with V have $\text{fst } ?M ! v = \text{bdd-map } (\lambda q. \text{the } (\text{bdd-lookup } bd \ q)) \ (\text{subsetbdd } (fst \ A) \ p \ (\text{nfa-emptybdd } (\text{length } p)))$ **by** simp
with well-formed BS P-len have $\text{bdd-lookup } (fst \ ?M ! v) \ \text{bs} = \text{the } (\text{bdd-lookup } bd \ (\text{bdd-lookup } (\text{subsetbdd } (fst \ A) \ p \ (\text{nfa-emptybdd } (\text{length } p)))) \ \text{bs})$
by (auto simp add: bdd-map-bdd-lookup bddh-subsetbdd wf-nfa-def is-alph-def nfa-emptybdd-def)
also from snoc J have $\dots = j$ **by** (simp add: nfa-trans-def)
finally have $JJ: \text{bdd-lookup } (fst \ ?M ! v) \ \text{bs} = j$.

from R BS JJ have $RR: \text{dfa-reach } ?M \ i \ (\text{bss } @ \ [\text{bs}]) \ j$ **by** (auto simp add: reach-snoc dfa-trans-def[symmetric])
with Z show ?case **by** simp
qed
next
assume $\text{dfa-reach } ?M \ i \ \text{bss } j \wedge \text{nfa-is-node } A \ q2$
hence $\text{dfa-reach } ?M \ i \ \text{bss } j$ **and** $\text{nfa-is-node } A \ q2$ **by** simp+
from this assms show $\text{nfa-reach } A \ q1 \ \text{bss } q2$
proof (induct arbitrary: q2)
case (snoc j bss bs q2)
define v **where** $v = \text{bdd-lookup } (fst \ ?M ! j) \ \text{bs}$
define qq **where** $qq = \text{nfa-trans } A \ (ls ! j) \ \text{bs}$
from well-formed have $Q0: \text{nfa-is-node } A \ (\text{nfa-startnode } A)$ **by** (simp add: nfa-startnode-is-node)

from snoc have $L: \text{length } (fst \ ?M) = \text{length } ls$ **by** (simp add: det-nfa-def)
with snoc have $\text{dfa-is-node } ?M \ i$ **by** (simp add: dfa-is-node-def) **moreover**
note $\langle \text{dfa-reach } ?M \ i \ \text{bss } j \rangle$ **moreover**
from snoc have $\text{wf-dfa } ?M \ n$ **by** (simp add: det-wf-nfa) **moreover**
from snoc have $\text{list-all } (\text{is-alph } n) \ \text{bss}$ **by** simp
ultimately have $\text{dfa-is-node } ?M \ j$ **by** (simp add: dfa-reach-is-node)
with L have $J\text{-len}: j < \text{length } ls$ **by** (simp add: dfa-is-node-def)

from $Q0$ **have** $list\text{-}all$ ($nfa\text{-}is\text{-}node$ A) (snd $?subset\text{-}dfs$) **by** ($rule$ $subset\text{-}dfs\text{-}is\text{-}node$)
with $snoc$ $J\text{-}len$ **have** J : $nfa\text{-}is\text{-}node$ A (ls ! j) **by** ($simp$ add : $list\text{-}all\text{-}iff$)
moreover **note** $snoc(4,5,6)$ $refl[of$ $ls!j]$ $snoc(8)$ $J\text{-}len$
moreover **from** $snoc$ **have** $list\text{-}all$ ($is\text{-}alph$ n) bss **by** $simp$
ultimately **have** R : $nfa\text{-}reach$ A $q1$ bss (ls ! j) **by** ($rule$ $snoc(2)$)
from $snoc$ **obtain** bs' **where** R' : $nfa\text{-}reach$ A ($nfa\text{-}startnode$ A) bs' $q1$ **and**
 BS' : $list\text{-}all$ ($is\text{-}alph$ n) bs' **by** $blast$
with R **have** lsj : $nfa\text{-}reach$ A ($nfa\text{-}startnode$ A) (bs' @ bss) (ls ! j) **by** ($simp$
 add : $reach\text{-}trans$)
hence $nfa\text{-}reach$ A ($nfa\text{-}startnode$ A) ((bs' @ bss) @ [bs]) qq **unfolding** $qq\text{-}def$
by ($rule$ $reach\text{-}snoc$)
with $well\text{-}formed$ $snoc(10)$ $Q0$ BS' **have** M : $subset\text{-}memb$ qq $?subset\text{-}dfs$ **and**
 $QQ\text{-}len$: $nfa\text{-}is\text{-}node$ A qq **by** ($simp$ add : $nfa\text{-}reach\text{-}subset\text{-}memb$ $nfa\text{-}reach\text{-}is\text{-}node$) +
with $snoc(4)$ **have** QQ : $bdd\text{-}lookup$ bd qq \neq $None$ **by** ($simp$ add : $subset\text{-}memb\text{-}def$)

from $well\text{-}formed$ $snoc$ J **have** H : $bddh$ ($length$ bs) ($subsetbdd$ (fst A) (ls ! j)
($nfa\text{-}emptybdd$ ($length$ (ls ! j)))) **by** ($simp$ add : $bddh\text{-}subsetbdd$ $wf\text{-}nfa\text{-}def$ $nfa\text{-}emptybdd\text{-}def$
 $is\text{-}alph\text{-}def$)
from $v\text{-}def$ **have** $v = bdd\text{-}lookup$ (fst $?M$! j) bs **by** $simp$
also **from** $snoc(4)$ **have** $\dots = bdd\text{-}lookup$ (map ($\lambda q. bdd\text{-}map$ ($\lambda q. the$ ($bdd\text{-}lookup$
 bd q)) ($subsetbdd$ (fst A) q ($nfa\text{-}emptybdd$ ($length$ q)))) (ls ! j) bs
by ($simp$ add : $det\text{-}nfa\text{-}def$)
also **from** $J\text{-}len$ **have** $\dots = bdd\text{-}lookup$ ($bdd\text{-}map$ ($\lambda q. the$ ($bdd\text{-}lookup$ bd q))
($subsetbdd$ (fst A) (ls ! j) ($nfa\text{-}emptybdd$ ($length$ (ls ! j)))))) bs **by** $simp$
also **from** H **have** $\dots = the$ ($bdd\text{-}lookup$ bd ($bdd\text{-}lookup$ ($subsetbdd$ (fst A) (ls
! j) ($nfa\text{-}emptybdd$ ($length$ (ls ! j)))))) bs) **by** ($simp$ add : $bdd\text{-}map\text{-}bdd\text{-}lookup$)
also **from** $qq\text{-}def$ **have** $\dots = the$ ($bdd\text{-}lookup$ bd qq) **by** ($simp$ add : $nfa\text{-}trans\text{-}def$)
finally **have** $v = the$ ($bdd\text{-}lookup$ bd qq) .
with QQ **have** QQ' : $bdd\text{-}lookup$ bd $qq = Some$ v **by** ($cases$ $bdd\text{-}lookup$ bd qq)
 $simp+$

with $snoc(4)$ **have** $bdd\text{-}lookup$ (fst $?subset\text{-}dfs$) $qq = Some$ v **by** $simp$
with $QQ\text{-}len$ $Q0$ **have** $v < length$ (snd $?subset\text{-}dfs$) \wedge (snd $?subset\text{-}dfs$) ! $v =$
 qq **by** ($simp$ add : $subset\text{-}dfs\text{-}bij$)
with $snoc$ $v\text{-}def$ **have** $Q2$: $qq = q2$ **by** ($simp$ add : $dfa\text{-}trans\text{-}def$)

with R $qq\text{-}def$ **show** $nfa\text{-}reach$ A $q1$ (bss @ [bs]) $q2$ **by** ($simp$ add : $reach\text{-}snoc$)
qed ($simp$ add : $reach\text{-}nil$)
qed

lemma $det\text{-}nfa\text{-}reach$:

fixes bd :: nat $option$ bdd **and** ls :: $bool$ $list$ $list$
assumes S : $subset\text{-}dfs$ A ($nfa\text{-}startnode$ A) = (bd , ls) (**is** $?subset\text{-}dfs = -$)
and $Q1$: $q1 = ls$! j **and** J : $j < length$ ls
and X : $list\text{-}all$ ($is\text{-}alph$ n) bss
shows $nfa\text{-}reach$ A ($nfa\text{-}startnode$ A) bss $q1 = dfa\text{-}reach$ ($det\text{-}nfa$ A) 0 bss j
proof –
note $SL = nfa\text{-}startnode\text{-}is\text{-}node[OF$ $well\text{-}formed]$
have $nfa\text{-}reach$ A ($nfa\text{-}startnode$ A) [] ($nfa\text{-}startnode$ A) **by** ($rule$ $reach\text{-}nil$)

hence 1: $\exists b. \text{nfa-reach } A \text{ (nfa-startnode } A) b \text{ (nfa-startnode } A) \wedge \text{list-all (is-alph } n) b$ **by** *auto*
from *SL* **have** $\text{bdd-lookup (fst ?subset-dfs) (nfa-startnode } A) = \text{Some } 0$ **by** (*simp add: subset-dfs-start*)
with *SL* **have** $0 < \text{length (snd ?subset-dfs)} \wedge \text{snd ?subset-dfs} \neq 0 = \text{nfa-startnode } A$ **by** (*simp add: subset-dfs-bij*)
with *S* **have** $2: 0 < \text{length } ls \wedge ls \neq 0 = \text{nfa-startnode } A$ **by** *simp*
from *S* 1 *Q1* *J* 2 *X* **have** $T: \text{nfa-reach } A \text{ (nfa-startnode } A) \text{ bss } q1 = (\text{dfa-reach (det-nfa } A) 0 \text{ bss } j \wedge \text{nfa-is-node } A \text{ } q1)$
by (*simp only: det-nfa-reach[^]*)
from *SL* **have** $\text{list-all (nfa-is-node } A) \text{ (snd ?subset-dfs)}$ **by** (*simp add: subset-dfs-is-node*)
with *Q1* *J* *S* **have** $\text{nfa-is-node } A \text{ } q1$ **by** (*simp add: list-all-iff*)
with *T* **show** *?thesis* **by** *simp*
qed

lemma *det-nfa-accepts:*

assumes *X: list-all (is-alph } n) w*
shows $\text{dfa-accepts (det-nfa } A) w = \text{nfa-accepts } A \text{ } w$
proof –
note $SL = \text{nfa-startnode-is-node}[OF \text{ well-formed}]$
let $?q = \text{nfa-startnode } A$
let $?subset-dfs = \text{subset-dfs } A \text{ (nfa-startnode } A)$
define *bd* **where** $bd = \text{fst ?subset-dfs}$
define *ls* **where** $ls = \text{snd ?subset-dfs}$
with *bd-def* **have** $BD: ?subset-dfs = (bd, ls)$ **by** *simp*
define *p* **where** $p = \text{nfa-steps } A \text{ (nfa-startnode } A) \text{ } w$
with *well-formed* *X* *SL* **have** $P: \text{nfa-is-node } A \text{ } p$ **by** (*simp add: nfa-steps-is-node*)
from *p-def* **have** $R: \text{nfa-reach } A \text{ } ?q \text{ } w \text{ } p$ **by** (*simp add: reach-def*)
with *assms* **have** $\exists bs. \text{nfa-reach } A \text{ } ?q \text{ } bs \text{ } p \wedge \text{list-all (is-alph } n) bs$ **by** *auto*
with *SL* **have** $(?q, p) \in (\text{succsr (subset-succs } A))^*$ **by** (*simp add: nfa-reach-rtrancl*)
with *SL* *P* **have** $\text{subset-memb } p \text{ } ?subset-dfs$ **by** (*simp add: dfs-eq-rtrancl*)
with *BD* **have** $\text{bdd-lookup } bd \text{ } p \neq \text{None}$ **by** (*simp add: subset-memb-def*)
then **obtain** *k* **where** $K: \text{bdd-lookup } bd \text{ } p = \text{Some } k$ **by** (*cases bdd-lookup } bd } p*)
simp+
with *SL* *P* **have** $K\text{-len: } k < \text{length } ls \wedge ls \neq k = p$ **unfolding** *bd-def* *ls-def* **by** (*simp add: subset-dfs-bij*)
with *BD* *X* *R* **have** $\text{dfa-reach (det-nfa } A) 0 \text{ } w \text{ } k$ **by** (*blast dest: det-nfa-reach*)
hence $k = \text{dfa-steps (det-nfa } A) 0 \text{ } w$ **by** (*simp add: reach-def*)
hence $\text{dfa-accepts (det-nfa } A) w = \text{snd (det-nfa } A) \text{ } ! k$ **by** (*simp add: accepts-def dfa-accepting-def*)
also **from** *ls-def* **have** $\dots = \text{map (nfa-accepting } A) ls \text{ } ! k$ **by** (*simp add: det-nfa-def split-beta*)
also **from** *K-len* *p-def* **have** $\dots = \text{nfa-accepts } A \text{ } w$ **by** (*simp add: accepts-def*)
finally **show** $\text{dfa-accepts (det-nfa } A) w = \text{nfa-accepts } A \text{ } w$.
qed

end

lemma *det-wf-nfa*:
assumes *A*: *wf-nfa A n*
shows *wf-dfa (det-nfa A) n*
proof –
from *A*
interpret *subset-DFS A n* **by** *unfold-locales*
show *?thesis* **by** (*rule det-wf-nfa*)
qed

lemma *det-nfa-accepts*:
assumes *A*: *wf-nfa A n*
and *w*: *list-all (is-alph n) bss*
shows *dfa-accepts (det-nfa A) bss = nfa-accepts A bss*
proof –
from *A*
interpret *subset-DFS A n* **by** *unfold-locales*
from *w* **show** *?thesis* **by** (*rule det-nfa-accepts*)
qed

5.5 Quantifiers

fun *quantify-bdd* :: *nat* \Rightarrow *bool list bdd* \Rightarrow *bool list bdd* **where**
quantify-bdd i (Leaf q) = Leaf q
| *quantify-bdd 0 (Branch l r) = (bdd-binop bv-or l r)*
| *quantify-bdd (Suc i) (Branch l r) = Branch (quantify-bdd i l) (quantify-bdd i r)*

lemma *bddh-quantify-bdd*:
assumes *bddh (Suc n) bdd* **and** *v* \leq *n*
shows *bddh n (quantify-bdd v bdd)*
using *assms* **by** (*induct v bdd arbitrary: n rule: quantify-bdd.induct*) (*auto simp: bddh-binop split: nat.splits*)

lemma *quantify-bdd-is-node*:
assumes *bdd-all (nfa-is-node N) bdd*
shows *bdd-all (nfa-is-node N) (quantify-bdd v bdd)*
using *assms* **by** (*induct v bdd rule: quantify-bdd.induct*) (*simp add: bdd-all-bdd-binop[of nfa-is-node N - nfa-is-node N - nfa-is-node N bv-or, OF - - bv-or-is-node]*)
qed

definition
quantify-nfa :: *nat* \Rightarrow *nfa* \Rightarrow *nfa* **where**
quantify-nfa i = (λ (bdds, as). (map (quantify-bdd i) bdds, as))

lemma *quantify-nfa-well-formed-aut*:
assumes *wf-nfa N (Suc n)*
and *v* \leq *n*
shows *wf-nfa (quantify-nfa v N) n*
proof –
from *assms* **have** 1: *list-all (bddh (Suc n)) (fst N)* **and** 2: *list-all (bdd-all (nfa-is-node N)) (fst N)* **by** (*simp add: wf-nfa-def*)
qed

from 1 assms have 3: $\text{list-all (bddh } n) (\text{fst (quantify-nfa } v \ N))$ **by** (*simp add: quantify-nfa-def bddh-quantify-bdd list-all-iff split-beta*)
from 2 have $\text{list-all (bdd-all (nfa-is-node } N)) (\text{fst (quantify-nfa } v \ N))$ **by** (*simp add: quantify-bdd-is-node list-all-iff split-beta quantify-nfa-def*)
hence $\text{list-all (bdd-all (nfa-is-node (quantify-nfa } v \ N))) (\text{fst (quantify-nfa } v \ N))$
by (*simp add: quantify-nfa-def split-beta nfa-is-node-def*)
with 3 assms show ?thesis **by** (*simp add: wf-nfa-def quantify-nfa-def split-beta*)
qed

fun $\text{insertl} :: \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$ **where**
 $\text{insertl } i \ a \ [] = [a]$
 $|\ \text{insertl } 0 \ a \ bs = a \ # \ bs$
 $|\ \text{insertl } (\text{Suc } i) \ a \ (b \ # \ bs) = b \ # \ (\text{insertl } i \ a \ bs)$

lemma *insertl-len:*
 $\text{length (insertl } n \ x \ vs) = \text{Suc (length } vs)$
by (*induct n x vs rule: insertl.induct*) *simp+*

lemma *insertl-0-eq:* $\text{insertl } 0 \ x \ xs = x \ # \ xs$
by (*cases xs*) *simp-all*

lemma *bdd-lookup-quantify-bdd-set-of-bv:*
assumes $\text{length } w = n$
and $\text{bddh (Suc } n) \ \text{bdd}$
and $\text{bdd-all (nfa-is-node } N) \ \text{bdd}$
and $v \leq n$
shows $\text{set-of-bv (bdd-lookup (quantify-bdd } v \ \text{bdd}) \ w) = (\bigcup b. \text{set-of-bv (bdd-lookup } \text{bdd (insertl } v \ b \ w))}$
using *assms proof* (*induct v bdd arbitrary: n w rule: quantify-bdd.induct*)
case ($2 \ l \ r \ w$)
hence $N: \text{nfa-is-node } N (\text{bdd-lookup } l \ w) \ \text{nfa-is-node } N (\text{bdd-lookup } r \ w)$ **by** (*simp add: bdd-all-bdd-lookup*)
have $\text{set-of-bv (bdd-lookup (quantify-bdd } 0 \ (\text{Branch } l \ r)) \ w) = \text{set-of-bv (bdd-lookup (bdd-binop } bv\text{-or } l \ r) \ w)}$ **by** *simp*
also from 2 have $\dots = \text{set-of-bv (bv-or (bdd-lookup } l \ w) (\text{bdd-lookup } r \ w))$ **by** (*simp add: bdd-lookup-binop*)
also from N have $\dots = \text{set-of-bv (bdd-lookup } l \ w) \cup \text{set-of-bv (bdd-lookup } r \ w)$
by (*simp add: bv-or-set-of-bv*)
also have $\dots = \text{set-of-bv (bdd-lookup (Branch } l \ r) (\text{insertl } 0 \ \text{False } w)) \cup \text{set-of-bv (bdd-lookup (Branch } l \ r) (\text{insertl } 0 \ \text{True } w))$ **by** (*cases w*) *simp+*
also have $\dots = (\bigcup b \in \{\text{True}, \text{False}\}. \text{set-of-bv (bdd-lookup (Branch } l \ r) (\text{insertl } 0 \ b \ w))}$ **by** *auto*
also have $\dots = (\bigcup b. \text{set-of-bv (bdd-lookup (Branch } l \ r) (\text{insertl } 0 \ b \ w))}$ **by** *blast*
finally show *?case .*
next
case ($3 \ n \ l \ r \ k \ w$)
then obtain j **where** $J: k = \text{Suc } j$ **by** (*cases k*) *simp+*
with 3 obtain a **as where** $W: w = a \ # \ as$ **by** (*cases w*) *auto*
with 3 J show *?case* **by** (*cases a*) *simp+*

qed *simp*

lemma *subsetbdd-set-of-bv*:

assumes *wf-nfa* N (*length* ws)
and *nfa-is-node* N q
shows *set-of-bv* (*bdd-lookup* (*subsetbdd* (*fst* N) q (*nfa-emptybdd* (*length* q))) ws)
 $=$ ($\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) ws)$)
(**is** *set-of-bv* $?q = -$)
proof (*simp only: set-eq-iff, rule allI*)
fix $x :: \text{nat}$
from *assms* **have** *bdd-all* (*nfa-is-node* N) (*subsetbdd* (*fst* N) q (*nfa-emptybdd* (*length* q)))
by (*simp add: wf-nfa-def bdd-all-is-node-subsetbdd*)
with *assms* **have** *nfa-is-node* N $?q$
by (*simp add: wf-nfa-def bdd-all-bdd-lookup bddh-subsetbdd nfa-emptybdd-def*)
hence L : *length* $?q = \text{length } (\text{fst } N)$ **by** (*simp add: nfa-is-node-def*)
{
fix i **assume** H : $i < \text{length } (\text{fst } N)$
with *assms* **have** *nfa-is-node* N (*bdd-lookup* (*fst* $N ! i$) ws) **by** (*simp add: wf-nfa-def list-all-iff bdd-all-bdd-lookup*)
}
with *assms* **have** I : $\bigwedge i. i < \text{length } q \implies \text{nfa-is-node } N (\text{bdd-lookup } (\text{fst } N ! i) ws)$ **by** (*simp add: nfa-is-node-def*)

from L *assms* **have** $x \in \text{set-of-bv } ?q = (x < \text{length } (\text{fst } N) \wedge (\exists i \in \text{set-of-bv } q. \text{bdd-lookup } (\text{fst } N ! i) ws ! x \wedge i < \text{length } q))$ **by** (*auto simp add: set-of-bv-def bdd-lookup-subsetbdd*)

also from I **have** $\dots = (x \in (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) ws)))$ **by** (*auto simp: nfa-is-node-def set-of-bv-def*)

finally show $x \in \text{set-of-bv } ?q = (x \in (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) ws)))$.

qed

lemma *nfa-trans-quantify-nfa*:

assumes *wf-nfa* N (*Suc* n)
and $v \leq n$
and *is-alph* n w
and *nfa-is-node* N q
shows *set-of-bv* (*nfa-trans* (*quantify-nfa* v N) q w) $=$ ($\bigcup b. \text{set-of-bv } (\text{nfa-trans } N q (\text{insertl } v b w))$)
proof –
from *assms* **have** $V1$: *wf-nfa* (*quantify-nfa* v N) n **by** (*simp add: quantify-nfa-well-formed-aut*)
with *assms* **have** $V2$: *wf-nfa* (*quantify-nfa* v N) (*length* w) **by** (*simp add: wf-nfa-def is-alph-def*)
from *assms* **have** N : *nfa-is-node* (*quantify-nfa* v N) q **by** (*simp add: quantify-nfa-def wf-nfa-def split-beta nfa-is-node-def*)
{ **fix** i **assume** H : $i \in \text{set-of-bv } q$
with *assms* **have** $i < \text{length } (\text{fst } N)$ **by** (*simp add: nfa-is-node-def set-of-bv-def*)
with *assms* **have** *bddh* (*Suc* n) (*fst* $N ! i$) *bdd-all* (*nfa-is-node* N) (*fst* $N ! i$)

by (*simp add: wf-nfa-def list-all-iff*)+
}
with *assms* **have** $I: \bigwedge i. i \in \text{set-of-bv } q \implies \text{length } w = n \wedge \text{bddh } (\text{Suc } n) (\text{fst } N ! i) \wedge \text{bdd-all } (\text{nfa-is-node } N) (\text{fst } N ! i) \wedge v \leq n$ **by** (*simp add: is-alph-def*)
from *assms* **have** $V3: \bigwedge b. \text{wf-nfa } N (\text{length } (\text{insertl } v b w))$ **by** (*simp add: wf-nfa-def is-alph-def insertl-len*)
from $N V2$ **have** $\text{set-of-bv } (\text{bdd-lookup } (\text{subsetbdd } (\text{fst } (\text{quantify-nfa } v N)) q (\text{nfa-emptybdd } (\text{length } q)))) w = (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{fst } (\text{quantify-nfa } v N) ! i) w))$
by (*simp add: subsetbdd-set-of-bv*)
also from *assms* **have** $\dots = (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v (\text{fst } N ! i) w)))$ **by** (*auto simp: quantify-nfa-def split-beta nfa-is-node-def set-of-bv-def*)
also have $\dots = (\bigcup_{i \in \text{set-of-bv } q} \bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w)))$
proof (*simp only: set-eq-iff, rule allI*)
fix x
have $x \in (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v (\text{fst } N ! i) w)))$
 $= (\exists i \in \text{set-of-bv } q. x \in \text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v (\text{fst } N ! i) w)))$ **by** *simp*
also have $\dots = (\{i. i \in \text{set-of-bv } q \wedge x \in \text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v (\text{fst } N ! i) w))\} \neq \{\})$ **by** *auto*
also from I **have** $\dots = (\{i. i \in \text{set-of-bv } q \wedge x \in (\bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w)))\} \neq \{\})$ **by** (*auto simp: bdd-lookup-quantify-bdd-set-of-bv[of w n - N]*)
also have $\dots = (\exists i \in \text{set-of-bv } q. x \in (\bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w))))$ **by** *auto*
also have $\dots = (x \in (\bigcup_{i \in \text{set-of-bv } q} \bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w))))$ **by** *simp*
finally show $(x \in (\bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v (\text{fst } N ! i) w)))) = (x \in (\bigcup_{i \in \text{set-of-bv } q} \bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w))))$.
qed
also have $\dots = (\bigcup b. \bigcup_{i \in \text{set-of-bv } q} \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N ! i) (\text{insertl } v b w)))$ **by** *auto*
also from $V3$ *assms* **have** $\dots = (\bigcup b. \text{set-of-bv } (\text{bdd-lookup } (\text{subsetbdd } (\text{fst } N) q (\text{nfa-emptybdd } (\text{length } q)))) (\text{insertl } v b w))$ **by** (*simp add: subsetbdd-set-of-bv*)
finally show *?thesis* **by** (*simp add: nfa-trans-def*)
qed

fun *insertll* :: *nat* \Rightarrow '*a list* \Rightarrow '*a list list* \Rightarrow '*a list list*

where

insertll i [] [] = []
| *insertll* i ($a \# as$) ($bs \# bss$) = *insertl* i a $bs \# \text{insertll } i$ as bss

lemma *insertll-len2*:

assumes *list-all* (*is-alph* n) *vs*
and *length* $x = \text{length } vs$
shows *list-all* (*is-alph* (*Suc* n)) (*insertll* k x vs)
using *assms* **by** (*induct* k x vs *rule: insertll.induct*) (*auto simp: insertll-len is-alph-def*)+

lemma *insertll-append*:

assumes $\text{length } xs = \text{length } vs$
shows $\text{insertll } k (xs @ [x]) (vs @ [v]) = \text{insertll } k xs vs @ [\text{insertl } k x v]$
using *assms* **by** (*induct k xs vs rule: insertll.induct*) *simp+*

lemma *UN-UN-lenset*: $(\bigcup b. \bigcup x \in \{x. \text{length } x = n\}. M b x) = (\bigcup bs \in \{x. \text{length } x = \text{Suc } n\}. M (\text{last } bs) (\text{butlast } bs))$

proof *auto*

fix $x b xa$ **assume** $x \in M b xa$
hence $\text{length } (xa @ [b]) = \text{Suc } (\text{length } xa) \wedge x \in M (\text{last } (xa @ [b])) (\text{butlast } (xa @ [b]))$ **by** *simp*
thus $\exists bs. \text{length } bs = \text{Suc } (\text{length } xa) \wedge x \in M (\text{last } bs) (\text{butlast } bs)$..
next
fix $x bs$ **assume** $x \in M (\text{last } bs) (\text{butlast } bs)$ **and** $\text{length } bs = \text{Suc } n$
hence $\text{length } (\text{butlast } bs) = n \wedge x \in M (\text{last } bs) (\text{butlast } bs)$ **by** *simp*
thus $\exists b xa. \text{length } xa = n \wedge x \in M b xa$ **by** *blast*

qed

lemma *nfa-steps-quantify-nfa*:

assumes *wf-nfa* $N (\text{Suc } n)$
and *list-all* (*is-alph* n) w
and *nfa-is-node* $N q$
and $v \leq n$
shows $\text{set-of-bv } (nfa\text{-steps } (quantify\text{-nfa } v N) q w) = (\bigcup xs \in \{x. \text{length } x = \text{length } w\}. \text{set-of-bv } (nfa\text{-steps } N q (\text{insertll } v xs w)))$
using *assms* **proof** (*induct w rule: rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc* $x xs$)
hence *wf-nfa* (*quantify-nfa* $v N$) n **by** (*simp add: quantify-nfa-well-formed-aut*)
moreover from *snoc* **have** *nfa-is-node* (*quantify-nfa* $v N$) q **by** (*simp add: nfa-is-node-def quantify-nfa-def split-beta*)
moreover note *snoc*
ultimately have *nfa-is-node* (*quantify-nfa* $v N$) (*nfa-steps* (*quantify-nfa* $v N$) $q xs$) **by** (*simp add: nfa-steps-is-node[of - n]*)
hence $N: nfa\text{-is-node } N (nfa\text{-steps } (quantify\text{-nfa } v N) q xs)$ (*is nfa-is-node* $N ?q$) **by** (*simp add: nfa-is-node-def quantify-nfa-def split-beta*)
from *snoc* **have** $\bigwedge b. \text{length } (\text{insertl } v b x) = \text{Suc } n$ **by** (*simp add: insertl-len is-alph-def*)
with *snoc* **have** $B: \bigwedge b. wf\text{-nfa } N (\text{length } (\text{insertl } v b x))$ **by** *simp*
from *snoc* **have** $IV: \text{set-of-bv } (nfa\text{-steps } (quantify\text{-nfa } v N) q xs) = (\bigcup x \in \{x. \text{length } x = \text{length } xs\}. \text{set-of-bv } (nfa\text{-steps } N q (\text{insertll } v x xs)))$ **by** *simp*

{ **fix** $bs :: \text{bool list}$ **assume** $H: \text{length } bs = \text{length } xs$
with *snoc* **have** *list-all* (*is-alph* ($\text{Suc } n$)) (*insertll* $v bs xs$) **by** (*simp add: insertll-len2*)
with *snoc* **have** *nfa-is-node* $N (nfa\text{-steps } N q (\text{insertll } v bs xs))$ **by** (*simp add: nfa-steps-is-node*)

} note $N2 = this$

have $set-of-bv (nfa-steps (quantify-nfa v N) q (xs @ [x])) = set-of-bv (nfa-steps (quantify-nfa v N) ?q [x])$

by *simp*

also have $\dots = set-of-bv (nfa-trans (quantify-nfa v N) ?q x)$ **by** *simp*

also from $snoc N$ **have** $\dots = (\bigcup b. set-of-bv (nfa-trans N ?q (insertl v b x)))$ **by** (*simp add: nfa-trans-quantify-nfa*)

also have $\dots = (\bigcup b. set-of-bv (bdd-lookup (subsetbdd (fst N) ?q (nfa-emptybdd (length ?q))) (insertl v b x)))$ **by** (*simp add: nfa-trans-def*)

also from $N B$ **have** $\dots = (\bigcup b. \bigcup i \in set-of-bv ?q. set-of-bv (bdd-lookup (fst N ! i) (insertl v b x)))$ **by** (*simp add: subsetbdd-set-of-bv*)

also from IV **have** $\dots = (\bigcup b. \bigcup i \in (\bigcup x \in \{x. length x = length xs\}. set-of-bv (nfa-steps N q (insertll v x xs))). set-of-bv (bdd-lookup (fst N ! i) (insertl v b x)))$ **by** *simp*

also have $\dots = (\bigcup b. \bigcup y \in \{x. length x = length xs\}. \bigcup i \in set-of-bv (nfa-steps N q (insertll v y xs)). set-of-bv (bdd-lookup (fst N ! i) (insertl v b x)))$ **by** *simp*

also have $\dots = (\bigcup bs \in \{x. length x = Suc (length xs)\}. \bigcup i \in set-of-bv (nfa-steps N q (insertll v (butlast bs) xs)). set-of-bv (bdd-lookup (fst N ! i) (insertl v (last bs) x)))$

by (*simp add: UN-UN-lenset*)

also from $N2 B$ **have** $\dots = (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-trans N (nfa-steps N q (insertll v (butlast bs) xs)) (insertl v (last bs) x)))$ (**is** $?L = ?R$)

by (*simp add: subsetbdd-set-of-bv[folded nfa-trans-def]*)

also have $\dots = (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v (butlast bs) xs @ [insertl v (last bs) x])))$

by *simp*

also have $\dots = (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v (butlast bs @ [last bs]) (xs @ [x])))$ **by** (*auto simp: insertll-append*)

also have $\dots = (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v bs (xs @ [x])))$

proof (*rule set-eqI*)

fix xa

have $(xa \in (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v (butlast bs @ [last bs]) (xs @ [x])))) =$

$(\exists bs \in \{x. length x = Suc (length xs)\}. bs \neq [] \wedge xa \in set-of-bv (nfa-steps N q (insertll v (butlast bs @ [last bs]) (xs @ [x])))$ **by** *auto*

also have $\dots = (\exists bs \in \{x. length x = Suc (length xs)\}. bs \neq [] \wedge xa \in set-of-bv (nfa-steps N q (insertll v bs (xs @ [x])))$ **by** *auto*

also have $\dots = (xa \in (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v bs (xs @ [x]))))$ **by** *auto*

finally show $(xa \in (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v (butlast bs @ [last bs]) (xs @ [x])))) =$

$(xa \in (\bigcup bs \in \{x. length x = Suc (length xs)\}. set-of-bv (nfa-steps N q (insertll v bs (xs @ [x]))))$.

qed

finally show $?case$ **by** *simp*

qed

lemma *nfa-accepts-quantify-nfa*:
assumes *wf-nfa A (Suc n)*
and $i \leq n$
and *list-all (is-alph n) bss*
shows $nfa\text{-accepts } (quantify\text{-nfa } i A) bss = (\exists bs. nfa\text{-accepts } A (insertll\ i\ bs\ bss))$
 $\wedge length\ bs = length\ bss$
proof –
note $Q0 = nfa\text{-startnode-is-node}[OF\ assms(1)]$
hence $nfa\text{-is-node } A (nfa\text{-startnode } (quantify\text{-nfa } i A))$ **by** (*simp add: nfa-startnode-def quantify-nfa-def split-beta*)
with *assms* **have** $I: set\text{-of-bv } (nfa\text{-steps } (quantify\text{-nfa } i A) (nfa\text{-startnode } (quantify\text{-nfa } i A)) bss) =$
 $(\bigcup bs \in \{bs. length\ bs = length\ bss\}. set\text{-of-bv } (nfa\text{-steps } A (nfa\text{-startnode } (quantify\text{-nfa } i A)) (insertll\ i\ bs\ bss)))$
by (*simp add: nfa-steps-quantify-nfa*)
have $nfa\text{-accepts } (quantify\text{-nfa } i A) bss = nfa\text{-accepting } (quantify\text{-nfa } i A) (nfa\text{-steps } (quantify\text{-nfa } i A) (nfa\text{-startnode } (quantify\text{-nfa } i A)) bss)$ **by** (*simp add: accepts-def*)
also **have** $\dots = (set\text{-of-bv } (snd\ (quantify\text{-nfa } i A)) \cap set\text{-of-bv } (nfa\text{-steps } (quantify\text{-nfa } i A) (nfa\text{-startnode } (quantify\text{-nfa } i A)) bss) \neq \{\})$ **by** (*simp add: nfa-accepting-set-of-bv*)
also **from** I **have** $\dots = (set\text{-of-bv } (snd\ A) \cap (\bigcup bs \in \{bs. length\ bs = length\ bss\}. set\text{-of-bv } (nfa\text{-steps } A (nfa\text{-startnode } (quantify\text{-nfa } i A)) (insertll\ i\ bs\ bss))) \neq \{\})$
by (*simp add: quantify-nfa-def split-beta*)
also **have** $\dots = ((\bigcup bs \in \{bs. length\ bs = length\ bss\}. set\text{-of-bv } (snd\ A) \cap set\text{-of-bv } (nfa\text{-steps } A (nfa\text{-startnode } (quantify\text{-nfa } i A)) (insertll\ i\ bs\ bss))) \neq \{\})$ **by** *simp*
also **have** $\dots = (\exists bs \in \{bs. length\ bs = length\ bss\}. set\text{-of-bv } (snd\ A) \cap set\text{-of-bv } (nfa\text{-steps } A (nfa\text{-startnode } A) (insertll\ i\ bs\ bss)) \neq \{\})$ **by** (*auto simp: nfa-startnode-def quantify-nfa-def split-beta*)
also **have** $\dots = (\exists bs. nfa\text{-accepts } A (insertll\ i\ bs\ bss) \wedge length\ bs = length\ bss)$
by (*auto simp: accepts-def nfa-accepting-set-of-bv*)
finally **show** *?thesis* .
qed

5.6 Right Quotient

definition

rquot-succs :: $nat\ bdd\ list \times bool\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list$ **where**
rquot-succs $M = (\lambda n\ x. [bdd\text{-lookup } (fst\ M\ !\ x) (replicate\ n\ False)])$

definition

rquot-invariant :: $nat\ bdd\ list \times bool\ list \Rightarrow bool\ list \Rightarrow bool$ **where**
rquot-invariant $M = (\lambda l. length\ l = length\ (fst\ M))$

definition

rquot-ins = $(\lambda x\ l. l[x:=True])$

definition

rquot-memb :: $nat \Rightarrow bool\ list \Rightarrow bool$ **where**

$rquot-memb = (\lambda x l. l ! x)$

definition

$rquot-empt :: nat \text{ bdd list} \times \text{ bool list} \Rightarrow \text{ bool list}$ **where**
 $rquot-empt M = replicate (length (fst M)) \text{ False}$

definition

$rquot-dfs M n x = gen-dfs (rquot-succs M n) rquot-ins rquot-memb (rquot-empt M) [x]$

definition

$zeros :: nat \Rightarrow nat \Rightarrow \text{ bool list list}$ **where**
 $zeros m n = replicate m (replicate n \text{ False})$

lemma zeros-is-alpha: $list-all (is-alph v) (zeros n v)$
by ($induct n$) ($simp \text{ add: zeros-def is-alph-def}$) $+$

lemma zeros-rone: $zeros (Suc n) v = zeros n v @ zeros 1 v$
by ($simp \text{ add: zeros-def replicate-append-same}$)

lemma zeros-len: $length (zeros n v) = n$
by ($simp \text{ add: zeros-def}$)

lemma zeros-rtrancl: $(\exists n. dfa-reach M x (zeros n v) y) = ((x,y) \in (succsr (rquot-succs M v))^*)$

proof

assume $\exists n. dfa-reach M x (zeros n v) y$
then obtain n **where** $N: dfa-reach M x (zeros n v) y$ **..**
define w **where** $w = zeros n v$
hence $W: \exists n. w = zeros n v$ **by auto**
from $w\text{-def } N$ **have** $dfa-reach M x w y$ **by simp**
from this W **show** $(x,y) \in (succsr (rquot-succs M v))^*$
proof induct
 case ($snoc k ws y$)
 then obtain n' **where** $N': ws @ [y] = zeros n' v$ **by blast**
 have $length (ws @ [y]) > 0$ **by simp**
 with N' **have** $n' > 0$ **by** ($simp \text{ add: zeros-len}$)
 then obtain n **where** $NL: n' = Suc n$ **by** ($cases n'$) $simp+$
 hence $zeros n' v = zeros n v @ zeros 1 v$ **by** ($simp \text{ only: zeros-rone}$)
 also have $\dots = zeros n v @ [replicate v \text{ False}]$ **by** ($simp \text{ add: zeros-def}$)
 finally have $zeros n' v = zeros n v @ [replicate v \text{ False}]$ **.**
 with N' **have** $WS: ws = zeros n v y = replicate v \text{ False}$ **by auto**
 hence $\exists n. ws = zeros n v$ **by auto**
 with $snoc$ **have** $IV: (x,k) \in (succsr (rquot-succs M v))^*$ **by simp**
 from WS **have** $dfa-trans M k y \in set (rquot-succs M v k)$ **by** ($simp \text{ add: rquot-succs-def dfa-trans-def}$)
 hence $(k, dfa-trans M k y) \in (succsr (rquot-succs M v))^*$ **by** ($auto \text{ simp: succsr-def}$)
 with IV **show** $?case$ **by simp**


```

qed simp
next
assume  $(x,y) \in (\text{succsr } (\text{rquot-succs } M v))^*$ 
thus  $\exists n. \text{dfa-reach } M x (\text{zeros } n v) y$ 
proof induct
  case base
    have  $\text{dfa-reach } M x (\text{zeros } 0 v) x$  by (simp add: reach-nil zeros-def)
    thus  $\exists n. \text{dfa-reach } M x (\text{zeros } n v) x$  by (rule exI)
  next
    case (step y z)
      then obtain  $n$  where  $N: \text{dfa-reach } M x (\text{zeros } n v) y$  by blast
      with step have  $Z: z = \text{dfa-trans } M y (\text{replicate } v \text{ False})$  by (simp add: succsr-def
rquot-succs-def dfa-trans-def)
      from  $N Z$  have  $\text{dfa-reach } M x (\text{zeros } n v @ \text{zeros } 1 v) z$  by (simp add: reach-snoc
zeros-def)
      hence  $\text{dfa-reach } M x (\text{zeros } (\text{Suc } n) v) z$  by (simp only: zeros-rone)
      thus ?case by (rule exI)
qed
qed

```

```

primrec map-index ::  $(\text{'a} \Rightarrow \text{nat} \Rightarrow \text{'b}) \Rightarrow \text{'a list} \Rightarrow \text{nat} \Rightarrow \text{'b list}$ 
where
  map-index f [] n = []
| map-index f (x#xs) n = f x n # map-index f xs (Suc n)

```

```

lemma map-index-len:
  length (map-index f ls n) = length ls
by (induct ls arbitrary: n) simp+

```

```

lemma map-index-nth:
  assumes  $i < \text{length } l$ 
  shows  $\text{map-index } f l n ! i = f (l ! i) (n + i)$ 
using assms proof (induct l arbitrary: n i)
  case (Cons a l n i)
    show ?case proof (cases i = 0)
      case False
        then obtain  $j$  where  $J: i = \text{Suc } j$  by (cases i) simp+
        with Cons show ?thesis by simp
      case True
        qed simp
    qed simp

```

```

definition
  rquot ::  $\text{dfa} \Rightarrow \text{nat} \Rightarrow \text{dfa}$  where
  rquot =  $(\lambda(\text{bd}, \text{as}) v. (\text{bd}, \text{map-index } (\lambda x n. \text{nfa-accepting}' \text{ as } (\text{rquot-dfs } (\text{bd}, \text{as}) v n)) \text{ as } 0))$ 

```

```

lemma rquot-well-formed-aut:
  assumes wf-dfa M n
  shows wf-dfa (rquot M n) n

```

using *assms* **by** (*simp add: rquot-def split-beta wf-dfa-def map-index-len dfa-is-node-def*)

lemma *rquot-node*:

dfa-is-node (*rquot M n*) *q* = *dfa-is-node M q*
by (*simp add: rquot-def dfa-is-node-def split-beta*)

lemma *rquot-steps*:

dfa-steps (*rquot M n*) *x w* = *dfa-steps M x w*
by (*simp add: rquot-def dfa-trans-def [abs-def] split-beta*)

locale *rquot-DFS* =

fixes *A* :: *dfa* **and** *n* :: *nat*
assumes *well-formed: wf-dfa A n*

sublocale *rquot-DFS* < *DFS rquot-succs A n dfa-is-node A*

rquot-invariant A rquot-ins rquot-memb rquot-empt A

proof (*insert well-formed, unfold-locales*)

fix *x y S* **assume** *dfa-is-node A x* **and** *dfa-is-node A y* **and** *rquot-invariant A S*
and \neg *rquot-memb y S*

thus *rquot-memb x (rquot-ins y S)* = (*x = y* \vee *rquot-memb x S*)

by (*cases x=y*) (*simp add: dfa-is-node-def rquot-invariant-def rquot-memb-def rquot-ins-def*)⁺

qed (*simp add: dfa-is-node-def rquot-memb-def rquot-empt-def*

rquot-succs-def rquot-invariant-def rquot-ins-def

bounded-nat-set-is-finite[of - length (fst A)]

dfa-trans-is-node[unfolded dfa-trans-def dfa-is-node-def is-alph-def])⁺

context *rquot-DFS*

begin

lemma *rquot-dfs-invariant*:

assumes *dfa-is-node A x*
shows *rquot-invariant A (rquot-dfs A n x)*
using *assms well-formed unfolding rquot-dfs-def*
by (*auto simp: dfs-invariant' empt-invariant*)

lemma *dfa-reach-rquot*:

assumes *dfa-is-node A x*
and *dfa-is-node A y*
shows *rquot-memb y (rquot-dfs A n x)* = ($\exists m. \textit{dfa-reach A x (zeros m n) y}$)

proof –

from *assms* **have** *rquot-memb y (rquot-dfs A n x)* = (*(x,y) ∈ (succsr (rquot-succs A n))**)

by (*simp add: dfs-eq-rtrancl rquot-dfs-def*)

also **have** $\dots = (\exists m. \textit{dfa-reach A x (zeros m n) y})$ **by** (*simp add: zeros-rtrancl*)
finally **show** *?thesis* .

qed

lemma *rquot-accepting*:

assumes *dfa-is-node* (*rquot A n*) *q*
shows *dfa-accepting* (*rquot A n*) *q* = ($\exists m$. *dfa-accepting A (dfa-steps A q (zeros m n))*)
proof –
from *assms* **have** *Q: dfa-is-node A q* **by** (*simp add: rquot-node*)
with *assms* **have** *rquot-invariant A (rquot-dfs A n q)* **by** (*simp add: rquot-dfs-invariant*)
hence *L: length (rquot-dfs A n q) = length (fst A)* **by** (*simp add: rquot-invariant-def*)

have *nfa-accepting' (snd A) (rquot-dfs A n q) = (set-of-bv (snd A) \cap set-of-bv (rquot-dfs A n q) \neq {})* **by** (*simp add: nfa-accepting'-set-of-bv*)
also **have** $\dots = (\exists i. i < \text{length} (\text{snd } A) \wedge \text{snd } A ! i \wedge i < \text{length} (\text{rquot-dfs } A \text{ n } q) \wedge \text{rquot-dfs } A \text{ n } q ! i)$ **by** (*auto simp: set-of-bv-def*)
also **from** *well-formed L* **have** $\dots = (\exists i. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge \text{rquot-memb } i (\text{rquot-dfs } A \text{ n } q))$ **by** (*auto simp add: wf-dfa-def dfa-is-node-def rquot-memb-def*)
also **have** $\dots = (\{i. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge \text{rquot-memb } i (\text{rquot-dfs } A \text{ n } q)\} \neq \{\})$ **by** *auto*
also **from** *assms Q* **have** $\dots = (\{i. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge (\exists m. \text{dfa-reach } A \text{ q } (\text{zeros } m \text{ n}) \text{ i})\} \neq \{\})$ **by** (*auto simp: dfa-reach-rquot*)
also **have** $\dots = (\{i. \exists m. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge i = \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n})\} \neq \{\})$ **by** (*simp add: reach-def*)
also **have** $\dots = (\exists i m. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge i = \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n}))$ **by** *auto*
also **have** $\dots = (\exists m. \text{snd } A ! \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n}))$
proof
assume $\exists m. \text{snd } A ! \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n})$
then **obtain** *m* **where** *N: snd A ! dfa-steps A q (zeros m n) ..*
from *well-formed Q zeros-is-alpha[of n m]* **have** *dfa-is-node A (dfa-steps A q (zeros m n))* **by** (*simp add: dfa-steps-is-node*)
with *N* **show** $\exists i m. \text{dfa-is-node } A \text{ i} \wedge \text{snd } A ! i \wedge i = \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n})$ **by** *auto*
qed *auto*
finally **have** *nfa-accepting' (snd A) (rquot-dfs A n q) = ($\exists m. \text{snd } A ! \text{dfa-steps } A \text{ q } (\text{zeros } m \text{ n})$)* .
with *well-formed assms* **show** *?thesis* **by** (*simp add: dfa-accepting-def rquot-def split-beta dfa-is-node-def map-index-nth wf-dfa-def*)
qed

end

lemma *rquot-accepts:*

assumes *A: wf-dfa A n*
and *list-all (is-alph n) bss*
shows *dfa-accepts (rquot A n) bss = ($\exists m. \text{dfa-accepts } A (\text{bss } @ \text{zeros } m \text{ n})$)*
proof –
from *A*
interpret *rquot-DFS A n* **by** *unfold-locales*
from *assms* **have** *V: wf-dfa (rquot A n) n* **by** (*simp add: rquot-well-formed-aut*)
hence *dfa-is-node (rquot A n) 0* **by** (*simp add: dfa-startnode-is-node*)
with *assms V* **have** *q: dfa-is-node (rquot A n) (dfa-steps (rquot A n) 0 bss)* **by**

(simp add: dfa-steps-is-node)

have $\text{dfa-accepts } (\text{rquot } A \ n) \ \text{bss} = \text{dfa-accepting } (\text{rquot } A \ n) \ (\text{dfa-steps } (\text{rquot } A \ n) \ 0 \ \text{bss})$ **by** (simp add: accepts-def)
also from $\text{assms } q$ **have** $\dots = (\exists m. \text{dfa-accepting } A \ (\text{dfa-steps } A \ (\text{dfa-steps } A \ 0 \ \text{bss}) \ (\text{zeros } m \ n)))$ **by** (simp add: rquot-accepting rquot-steps)
also have $\dots = (\exists m. \text{dfa-accepting } A \ (\text{dfa-steps } A \ 0 \ (\text{bss} \ @ \ \text{zeros } m \ n)))$ **by** simp
also have $\dots = (\exists m. \text{dfa-accepts } A \ (\text{bss} \ @ \ \text{zeros } m \ n))$ **by** (simp add: accepts-def)
finally show $?thesis$.
qed

5.7 Diophantine Equations

fun $\text{eval-dioph} :: \text{int list} \Rightarrow \text{nat list} \Rightarrow \text{int}$

where

$\text{eval-dioph } (k \ \# \ ks) \ (x \ \# \ xs) = k * \text{int } x + \text{eval-dioph } ks \ xs$
 $| \text{eval-dioph } ks \ xs = 0$

lemma eval-dioph-mult :

$\text{eval-dioph } ks \ xs * \text{int } n = \text{eval-dioph } ks \ (\text{map } (\lambda x. x * n) \ xs)$
by (induct ks xs rule: eval-dioph.induct) (simp-all add: distrib-right)

lemma $\text{eval-dioph-add-map}$:

$\text{eval-dioph } ks \ (\text{map } f \ xs) + \text{eval-dioph } ks \ (\text{map } g \ xs) =$
 $\text{eval-dioph } ks \ (\text{map } (\lambda x. f \ x + g \ x) \ (xs :: \text{nat list}))$

proof (induct ks xs rule: eval-dioph.induct)

case (1 k ks x xs)

have $\text{eval-dioph } (k \ \# \ ks) \ (\text{map } f \ (x \ \# \ xs)) + \text{eval-dioph } (k \ \# \ ks) \ (\text{map } g \ (x \ \# \ xs)) =$

$(k * \text{int } (f \ x) + k * \text{int } (g \ x)) + (\text{eval-dioph } ks \ (\text{map } f \ xs) + \text{eval-dioph } ks \ (\text{map } g \ xs))$

by simp

also have $\dots = (k * \text{int } (f \ x) + k * \text{int } (g \ x)) + \text{eval-dioph } ks \ (\text{map } (\lambda x. f \ x + g \ x) \ xs)$

by (simp add: 1)

finally show $?case$ **by** (simp add: ac-simps distrib-left)

qed simp-all

lemma $\text{eval-dioph-div-mult}$:

$\text{eval-dioph } ks \ (\text{map } (\lambda x. x \ \text{div } n) \ xs) * \text{int } n +$
 $\text{eval-dioph } ks \ (\text{map } (\lambda x. x \ \text{mod } n) \ xs) = \text{eval-dioph } ks \ xs$
by (simp add: eval-dioph-mult o-def eval-dioph-add-map)

lemma eval-dioph-mod :

$\text{eval-dioph } ks \ xs \ \text{mod } \text{int } n = \text{eval-dioph } ks \ (\text{map } (\lambda x. x \ \text{mod } n) \ xs) \ \text{mod } \text{int } n$

proof (induct ks xs rule: eval-dioph.induct)

case (1 k ks x xs)

have $\text{eval-dioph } (k \ \# \ ks) \ (x \ \# \ xs) \ \text{mod } \text{int } n =$

$((k * \text{int } x) \text{ mod } \text{int } n + \text{eval-dioph } ks \text{ } xs \text{ mod } \text{int } n) \text{ mod } \text{int } n$
by (*simp add: mod-add-eq*)
also have $\dots = ((k * (\text{int } x \text{ mod } \text{int } n)) \text{ mod } \text{int } n +$
 $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } n) xs) \text{ mod } \text{int } n) \text{ mod } \text{int } n$
by (*simp add: 1 mod-mult-right-eq*)
finally show *?case* **by** (*simp add: zmod-int mod-add-eq*)
qed *simp-all*

lemma *eval-dioph-div-mod*:

$(\text{eval-dioph } ks \text{ } xs = l) =$
 $(\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs) \text{ mod } 2 = l \text{ mod } 2 \wedge$
 $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ div } 2) xs) =$
 $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ div } 2) \text{ (is } ?l = ?r)$

proof

assume *eq*: *?l*
then have $\text{eval-dioph } ks \text{ } xs \text{ mod } 2 = l \text{ mod } 2$ **by** *simp*
with *eval-dioph-mod* [*of - - 2*]
have *eq'*: $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs) \text{ mod } 2 = l \text{ mod } 2$
by *simp*
from *eval-dioph-div-mult* [*symmetric, of ks xs 2*] *eq*
have $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ div } 2) xs) * 2 + \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs) = l$
by *simp*
then have $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ div } 2) xs) * 2 = l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)$
by (*simp add: eq-diff-eq*)
then have $(\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ div } 2) xs) * 2) \text{ div } 2 =$
 $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ div } 2$
by *simp*
with *eq'* **show** *?r* **by** *simp*
next
assume *?r* (is *?r1* \wedge *?r2*)
then obtain *eq1*: *?r1* **and** *eq2*: *?r2* ..
from *eq1* **have** $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs) \text{ mod } 2) \text{ mod } 2 =$
 $(l - l \text{ mod } 2) \text{ mod } 2$
by *simp*
then have $(l \text{ mod } 2 - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs) \text{ mod } 2 \text{ mod } 2) \text{ mod } 2 =$
 $(l \text{ mod } 2 - l \text{ mod } 2 \text{ mod } 2) \text{ mod } 2$
by (*simp only: mod-diff-eq*)
then have *eq1'*: $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ mod } 2 = 0$
by (*simp add: mod-diff-eq*)
from *eq2* **have**
 $\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ div } 2) xs) * 2 +$
 $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ mod } 2 =$
 $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ div } 2 * 2 +$
 $(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ mod } 2$
by *simp*
then have

$eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2 +$
 $(l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ mod\ 2 =$
 $l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs)$
 by *simp*
 with *eq1' eval-dioph-div-mult [of - 2]* show ?l
 by (*simp add: eq-diff-eq*)
qed

lemma *eval-dioph-ineq-div-mod:*

$(eval_dioph\ ks\ xs \leq l) =$
 $(eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) \leq$
 $(l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ div\ 2) \text{ (is } ?l = ?r)$

proof

assume ?l
 with *eval-dioph-div-mult [symmetric, of ks xs 2]*
 have $eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2 + eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs) \leq l$
 by *simp*
 then have $eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2 \leq l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs)$
 by (*simp add: le-diff-eq*)
 then have $(eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2)\ div\ 2 \leq (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ div\ 2$
 by (*rule zdiv-mono1*) *simp*
 then show ?r by *simp*

next

assume ?r
 have $eval_dioph\ ks\ xs \leq eval_dioph\ ks\ xs + (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ mod\ 2$
 by *simp*
 also {
 from <?r> have $eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2 \leq (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ div\ 2 * 2$
 by *simp*
 also have $\dots = l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs) - (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ mod\ 2$
 by (*simp add: eq-diff-eq*)
 finally have $(eval_dioph\ ks\ (map\ (\lambda x. x\ div\ 2)\ xs) * 2 + eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs)) + (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ mod\ 2 \leq l$
 by *simp*
 with *eval-dioph-div-mult [of - 2]*
 have $eval_dioph\ ks\ xs + (l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs))\ mod\ 2 \leq l$
 by *simp* }
 finally show ?l .

qed

lemma *sum-list-abs-ge-0:* $(0::int) \leq sum_list\ (map\ abs\ ks)$

by (induct ks) simp-all

lemma *zmult-div-aux1*:

assumes $b: b \neq 0$

shows $(a - a \bmod b) \operatorname{div} b = (a::\operatorname{int}) \operatorname{div} b$

proof –

from *minus-mod-eq-mult-div* [*symmetric, of b a*]

have $(b * (a \operatorname{div} b)) \operatorname{div} b = (a - a \bmod b) \operatorname{div} b$

by *simp*

with b show *?thesis* by *simp*

qed

lemma *zmult-div-aux2*:

assumes $b: b \neq 0$

shows $((a::\operatorname{int}) - a \bmod b) \bmod b = 0$

using b *minus-mod-eq-mult-div* [*symmetric, of b a, symmetric*]

by *simp*

lemma *div-abs-eq*:

assumes *mod*: $(a::\operatorname{int}) \bmod b = 0$

and $b: 0 < b$

shows $|a \operatorname{div} b| = |a| \operatorname{div} b$

proof (*cases* $0 \leq a$)

case *True* with *pos-imp-zdiv-nonneg-iff* [*OF b*]

show *?thesis* by *auto*

next

from b have $b \neq 0$ by *auto*

case *False*

then have $a < 0$ by *auto*

have $|a \operatorname{div} b| = -(a \operatorname{div} b)$

by (*simp add: div-neg-pos-less0* [*OF* $\langle a < 0 \rangle b$] *zabs-def*)

with *abs-of-neg* [*OF* $\langle a < 0 \rangle$] *zdiv-zminus1-eq-if* [*OF* $\langle b \neq 0 \rangle$] *mod*

show *?thesis* by *simp*

qed

lemma *add-div-trivial*: $0 \leq c \implies c < b \implies ((a::\operatorname{int}) * b + c) \operatorname{div} b = a$

by (*simp add: div-add1-eq div-pos-pos-trivial*)

lemma *dioph-rhs-bound*:

$|(l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs})) \operatorname{div} 2| \leq \max |l| (\sum k \leftarrow \text{ks}. |k|)$

proof –

have $|(l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs})) \operatorname{div} 2| =$

$|(l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs}) -$

$(l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs})) \bmod 2) \operatorname{div} 2|$

(*is - = |(- - ?r) div 2|*)

by (*simp add: zmult-div-aux1*)

also have $\dots = |l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs}) - ?r| \operatorname{div} 2$

by (*simp add: zmult-div-aux2 div-abs-eq*)

also have $|l - \operatorname{eval-dioph} \text{ ks } (\operatorname{map} (\lambda x. x \bmod 2) \text{ xs}) - ?r| \leq$

$|l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)| + |?r|$
by (*rule abs-triangle-ineq4*)
also have $|l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)| \leq$
 $|l| + |\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)|$
by (*rule abs-triangle-ineq4*)
also have $|\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)| \leq (\sum k \leftarrow ks. |k|)$
proof (*induct ks xs rule: eval-dioph.induct*)
case (1 *k ks x xs*)
have $|k * \text{int } (x \text{ mod } 2) + \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)| \leq$
 $|k * \text{int } (x \text{ mod } 2)| + |\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)|$
by (*rule abs-triangle-ineq*)
also have $|k * \text{int } (x \text{ mod } 2)| \leq |k| * |\text{int } (x \text{ mod } 2)|$
by (*simp add: abs-mult*)
also have $|\text{int } (x \text{ mod } 2)| \leq 1$ **by** *simp*
finally have $|k * \text{int } (x \text{ mod } 2) + \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)| \leq$
 $|k| + |\text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)|$
by (*auto simp add: mult-left-mono*)
with 1 show *?case* **by** *simp*
qed (*simp-all add: sum-list-abs-ge-0*)
finally have *ineq*: $|(l - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ div } 2| \leq$
 $(|l| + (\sum k \leftarrow ks. |k|) + |?r|) \text{ div } 2$
by (*simp add: zdiv-mono1*)
show *?thesis*
proof (*cases* $(\sum k \leftarrow ks. |k|) \leq |l|$)
case *True*
note *ineq*
also from *True*
have $(|l| + (\sum k \leftarrow ks. |k|) + |?r|) \text{ div } 2 \leq (|l| * 2 + |?r|) \text{ div } 2$
by (*simp add: zdiv-mono1*)
also have $\dots = |l|$
by (*simp add: add-div-trivial*)
finally show *?thesis* **by** *simp*
next
case *False*
note *ineq*
also from *False*
have $(|l| + (\sum k \leftarrow ks. |k|) + |?r|) \text{ div } 2 \leq ((\sum k \leftarrow ks. |k|) * 2 + |?r|) \text{ div } 2$
by (*simp add: zdiv-mono1*)
also have $\dots = (\sum k \leftarrow ks. |k|)$
by (*simp add: add-div-trivial*)
finally show *?thesis* **by** *simp*
qed
qed

lemma *dioph-rhs-invariant*:
assumes *m*: $|m| \leq \max |l| (\sum k \leftarrow ks. |k|)$
shows $|(m - \text{eval-dioph } ks (\text{map } (\lambda x. x \text{ mod } 2) xs)) \text{ div } 2| \leq \max |l| (\sum k \leftarrow ks. |k|)$
proof (*cases* $(\sum k \leftarrow ks. |k|) \leq |l|$)


```

case True
have  $|(m - \text{eval-dioph } ks (\text{map } (\lambda x. x \bmod 2) xs)) \text{ div } 2| \leq \max |m| (\sum k \leftarrow ks. |k|)$ 
  by (rule dioph-rhs-bound)
also from True m have  $|m| \leq |l|$  by simp
finally show ?thesis by simp
next
case False
have  $|(m - \text{eval-dioph } ks (\text{map } (\lambda x. x \bmod 2) xs)) \text{ div } 2| \leq \max |m| (\sum k \leftarrow ks. |k|)$ 
  by (rule dioph-rhs-bound)
also from False m have  $|m| \leq (\sum k \leftarrow ks. |k|)$  by simp
also have  $\max (\sum k \leftarrow ks. |k|) (\sum k \leftarrow ks. |k|) \leq \max |l| (\sum k \leftarrow ks. |k|)$ 
  by simp
finally show ?thesis by simp
qed

```

lemma *bounded-int-set-is-finite*:

assumes *S*: $\forall (i::\text{int}) \in S. |i| < j$

shows *finite S*

proof (*rule finite-subset*)

have *finite* (*int* ‘ $\{n. n < \text{nat } j\}$)

by (*rule nat-seg-image-imp-finite [OF refl]*)

moreover have *finite* (($\lambda n. - \text{int } n$) ‘ $\{n. n < \text{nat } j\}$)

by (*rule nat-seg-image-imp-finite [OF refl]*)

ultimately show *finite* (*int* ‘ $\{n. n < \text{nat } j\} \cup (\lambda n. - \text{int } n)$ ‘ $\{n. n < \text{nat } j\}$)

by (*rule finite-UnI*)

show $S \subseteq \text{int } ‘ \{n. n < \text{nat } j\} \cup (\lambda n. - \text{int } n)$ ‘ $\{n. n < \text{nat } j\}$

proof

fix *i*

assume *i*: $i \in S$

show $i \in \text{int } ‘ \{n. n < \text{nat } j\} \cup (\lambda n. - \text{int } n)$ ‘ $\{n. n < \text{nat } j\}$

proof (*cases* $0 \leq i$)

case *True*

then have $i = \text{int } (\text{nat } i)$ **by** *simp*

moreover from *i S* **have** $\text{nat } i \in \{n. n < \text{nat } j\}$

by *auto*

ultimately have $i \in \text{int } ‘ \{n. n < \text{nat } j\}$

by (*rule image-eqI*)

then show *?thesis* ..

next

case *False*

then have $i = - \text{int } (\text{nat } (- i))$ **by** *simp*

moreover from *i S* **have** $\text{nat } (- i) \in \{n. n < \text{nat } j\}$

by *auto*

ultimately have $i \in (\lambda n. - \text{int } n)$ ‘ $\{n. n < \text{nat } j\}$

by (*rule image-eqI*)

then show *?thesis* ..

qed

qed
qed

primrec *mk-nat-vecs* :: *nat* \Rightarrow *nat list list* **where**

mk-nat-vecs 0 = []
| *mk-nat-vecs* (*Suc* *n*) =
 (*let* *yss* = *mk-nat-vecs* *n*
 in *map* (*Cons* 0) *yss* @ *map* (*Cons* 1) *yss*)

lemma *mk-nat-vecs-bound*: $\forall xs \in \text{set } (mk\text{-nat-vecs } n). \forall x \in \text{set } xs. x < 2$
by (*induct* *n*) (*auto simp add: Let-def*)

lemma *mk-nat-vecs-mod-eq*: $xs \in \text{set } (mk\text{-nat-vecs } n) \Longrightarrow \text{map } (\lambda x. x \bmod 2) xs = xs$

apply (*drule* *bspec* [*OF mk-nat-vecs-bound*])
apply (*induct* *xs*)
apply *simp-all*
done

definition

dioph-succs *n ks m* = *List.map-filter* ($\lambda xs.$
 if *eval-dioph* *ks xs mod* 2 = *m mod* 2
 then *Some* ((*m* - *eval-dioph* *ks xs*) *div* 2)
 else *None*) (*mk-nat-vecs* *n*)

definition

dioph-is-node :: *int list* \Rightarrow *int* \Rightarrow *int* \Rightarrow *bool* **where**
dioph-is-node *ks l m* = ($|m| \leq \max |l| (\sum k \leftarrow ks. |k|)$)

definition

dioph-invariant :: *int list* \Rightarrow *int* \Rightarrow *nat option list* \times *int list* \Rightarrow *bool* **where**
dioph-invariant *ks l* = ($\lambda(is, js). \text{length } is = \text{nat } (2 * \max |l| (\sum k \leftarrow ks. |k|) + 1)$)

definition

dioph-ins *m* = ($\lambda(is, js). (is[\text{int-encode } m := \text{Some } (\text{length } js)], js @ [m])$)

definition

dioph-memb :: *int* \Rightarrow *nat option list* \times *int list* \Rightarrow *bool* **where**
dioph-memb *m* = ($\lambda(is, js). is ! \text{int-encode } m \neq \text{None}$)

definition

dioph-empt :: *int list* \Rightarrow *int* \Rightarrow *nat option list* \times *int list* **where**
dioph-empt *ks l* = (*replicate* ($\text{nat } (2 * \max |l| (\sum k \leftarrow ks. |k|) + 1)$) *None*, [])

lemma *int-encode-bound*: *dioph-is-node* *ks l m* \Longrightarrow

int-encode *m* < $\text{nat } (2 * \max |l| (\sum k \leftarrow ks. |k|) + 1)$

by (*simp add: dioph-is-node-def int-encode-def sum-encode-def*) *arith*

```

interpretation dioph-dfs: DFS dioph-succs n ks dioph-is-node ks l
  dioph-invariant ks l dioph-ins dioph-memb dioph-empt ks l
proof (standard, goal-cases)
  case (1 x y)
  then show ?case
    apply (simp add: dioph-memb-def dioph-ins-def split-beta dioph-invariant-def)
    apply (cases x = y)
    apply (simp add: int-encode-bound)
    apply (simp add: inj-eq [OF inj-int-encode])
    done
  next
  case 2
  then show ?case
    by (simp add: dioph-memb-def dioph-empt-def int-encode-bound)
  next
  case 3
  then show ?case
    apply (simp add: dioph-succs-def map-filter-def list-all-iff dioph-is-node-def)
    apply (rule allI impI)+
    apply (erule subst [OF mk-nat-vecs-mod-eq])
    apply (drule dioph-rhs-invariant)
    apply assumption
    done
  next
  case 4
  then show ?case
    by (simp add: dioph-invariant-def dioph-empt-def)
  next
  case 5
  then show ?case
    by (simp add: dioph-invariant-def dioph-ins-def split-beta)
  next
  case 6
  then show ?case
    apply (rule bounded-int-set-is-finite [of - max |l| (∑ k←ks. |k|) + 1])
    apply (rule ballI)
    apply (simp add: dioph-is-node-def)
    done
qed

```

definition

dioph-dfs *n ks l* = *gen-dfs* (*dioph-succs* *n ks*) *dioph-ins* *dioph-memb* (*dioph-empt* *ks l*) [*l*]

primrec *make-bdd* :: (*nat list* ⇒ '*a*) ⇒ *nat* ⇒ *nat list* ⇒ '*a* *bdd*

where

make-bdd *f* 0 *xs* = *Leaf* (*f xs*)
 | *make-bdd* *f* (*Suc* *n*) *xs* = *Branch* (*make-bdd* *f* *n* (*xs* @ [0])) (*make-bdd* *f* *n* (*xs* @ [1]))

definition

```

eq-dfa n ks l =
  (let (is, js) = dioph-dfs n ks l
    in
    (map (λj. make-bdd (λxs.
      if eval-dioph ks xs mod 2 = j mod 2
      then the (is ! int-encode ((j - eval-dioph ks xs) div 2))
      else length js) n []) js @ [Leaf (length js)],
    map (λj. j = 0) js @ [False]))

```

abbreviation $(input) \text{ nat-of-bool} :: \text{bool} \Rightarrow \text{nat}$ **where** $\text{nat-of-bool} \equiv \text{of-bool}$ **lemma** $\text{nat-of-bool-bound}: \text{nat-of-bool } b < 2$ **by** $(\text{cases } b) \text{ simp-all}$ **lemma** $\text{nat-of-bool-mk-nat-vecs}$: $\text{length } bs = n \implies \text{map nat-of-bool } bs \in \text{set } (\text{mk-nat-vecs } n)$ **apply** $(\text{induct } n \text{ arbitrary: } bs)$ **apply** simp **apply** $(\text{case-tac } bs)$ **apply** simp **apply** $(\text{case-tac } a)$ **apply** $(\text{simp-all add: Let-def})$ **done****lemma** $\text{bdd-lookup-make-bdd}$: $\text{length } bs = n \implies \text{bdd-lookup } (\text{make-bdd } f \ n \ xs) \ bs = f \ (xs \ @ \ \text{map nat-of-bool } bs)$ **apply** $(\text{induct } n \text{ arbitrary: } bs \ xs)$ **apply** simp **apply** $(\text{case-tac } bs)$ **apply** auto **done****primrec** $\text{nat-of-bools} :: \text{bool list} \Rightarrow \text{nat}$ **where** $\text{nat-of-bools } [] = 0$ $| \text{nat-of-bools } (b \ # \ bs) = \text{nat-of-bool } b + 2 * \text{nat-of-bools } bs$ **primrec** $\text{nats-of-boolss} :: \text{nat} \Rightarrow \text{bool list list} \Rightarrow \text{nat list}$ **where** $\text{Nil: nats-of-boolss } n \ [] = \text{replicate } n \ 0$ $| \text{Cons: nats-of-boolss } n \ (bs \ # \ bss) =$ $\text{map } (\lambda(b, x). \text{nat-of-bool } b + 2 * x) \ (\text{zip } bs \ (\text{nats-of-boolss } n \ bss))$ **lemma** $\text{nats-of-boolss-length}$: $\text{list-all } (\text{is-alph } n) \ bss \implies \text{length } (\text{nats-of-boolss } n \ bss) = n$

by (induct bss) (simp-all add: is-alph-def)

lemma nats-of-boolss-mod2:

assumes bs : $\text{length } bs = n$ and bss : $\text{list-all } (is\text{-alph } n) bss$

shows $\text{map } (\lambda x. x \bmod 2) (\text{nats-of-boolss } n (bs \# bss)) = \text{map nat-of-bool } bs$

proof –

from bs bss

have $\text{map nat-of-bool } (\text{map fst } (\text{zip } bs (\text{nats-of-boolss } n bss))) = \text{map nat-of-bool } bs$

by (simp add: nats-of-boolss-length)

then show ?thesis

by (simp add: split-def o-def nat-of-bool-bound)

qed

lemma nats-of-boolss-div2:

assumes bs : $\text{length } bs = n$ and bss : $\text{list-all } (is\text{-alph } n) bss$

shows $\text{map } (\lambda x. x \text{ div } 2) (\text{nats-of-boolss } n (bs \# bss)) = \text{nats-of-boolss } n bss$

using bs bss

by (simp add: split-def o-def nat-of-bool-bound nats-of-boolss-length)

lemma zip-insertl: $\text{length } xs = \text{length } ys \implies$

$\text{zip } (\text{insertl } n x xs) (\text{insertl } n y ys) = \text{insertl } n (x, y) (\text{zip } xs ys)$

by (induct n x xs arbitrary: ys rule: insertl.induct)

(auto simp add: Suc-length-conv)

lemma map-insertl: $\text{map } f (\text{insertl } i x xs) = \text{insertl } i (f x) (\text{map } f xs)$

by (induct i x xs rule: insertl.induct) simp-all

lemma insertl-replicate: $m \leq n \implies$

$\text{insertl } m x (\text{replicate } n x) = x \# \text{replicate } n x$

apply (induct n arbitrary: m)

apply simp

apply (case-tac m)

apply simp-all

done

lemma nats-of-boolss-insertll:

$\text{list-all } (is\text{-alph } n) bss \implies \text{length } bs = \text{length } bss \implies i \leq n \implies$

$\text{nats-of-boolss } (Suc\ n) (\text{insertll } i bs bss) = \text{insertl } i (\text{nat-of-bools } bs) (\text{nats-of-boolss } n bss)$

by (induct i bs bss rule: insertll.induct)

(simp-all add: zip-insertl nats-of-boolss-length insertll-len2 is-alph-def

map-insertl insertl-replicate cong: conj-cong)

lemma zip-replicate-map: $\text{length } xs = n \implies \text{zip } (\text{replicate } n x) xs = \text{map } (\text{Pair } x) xs$

apply (induct n arbitrary: xs)

apply simp

apply (case-tac xs)

apply *simp-all*
done

lemma *zip-replicate-mapr*: $\text{length } xs = n \implies \text{zip } xs (\text{replicate } n x) = \text{map } (\lambda y. (y, x)) xs$
apply (*induct n arbitrary: xs*)
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

lemma *zip-assoc*: $\text{map } f (\text{zip } xs (\text{zip } ys zs)) = \text{map } (\lambda((x, y), z). f (x, (y, z))) (\text{zip } (\text{zip } xs ys) zs)$
apply (*induct xs arbitrary: ys zs*)
apply *simp*
apply (*case-tac ys*)
apply *simp*
apply (*case-tac zs*)
apply *simp-all*
done

lemma *nats-of-boolss-append*:
 $\text{list-all } (is\text{-alph } n) bss \implies \text{list-all } (is\text{-alph } n) bss' \implies$
 $\text{nats-of-boolss } n (bss @ bss') =$
 $\text{map } (\lambda(x, y). x + 2^{\text{length } bss} * y) (\text{zip } (\text{nats-of-boolss } n bss) (\text{nats-of-boolss } n bss'))$
by (*induct bss*)
(auto simp add: nats-of-boolss-length zip-replicate-map o-def map-zip-map map-zip-map2 zip-assoc is-alph-def)

lemma *nats-of-boolss-zeros*: $\text{nats-of-boolss } n (\text{zeros } m n) = \text{replicate } n 0$
by (*induct m*) (*simp-all add: zeros-def*)

declare *nats-of-boolss.Cons* [*simp del*]

fun *bools-of-nat* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool list}$

where

$\text{bools-of-nat } k n =$
(if $n = 0$ *then*
(if $k = 0$ *then* \square *else* $\text{False} \# \text{bools-of-nat } (k - 1) n$
else $(n \bmod 2 = 1) \# \text{bools-of-nat } (k - 1) (n \text{ div } 2)$ *)*

lemma *bools-of-nat-length*: $k \leq \text{length } (\text{bools-of-nat } k n)$
apply (*induct k n rule: bools-of-nat.induct*)
apply (*case-tac n = 0*)
apply (*case-tac k = 0*)
apply *simp*
apply *simp*
apply (*subst bools-of-nat.simps*)

```

apply (simp del: bools-of-nat.simps)
done

lemma nat-of-bool-mod-eq: nat-of-bool (n mod 2 = 1) = n mod 2
by (cases n mod 2 = 1) simp-all

lemma bools-of-nat-inverse: nat-of-bools (bools-of-nat k n) = n
apply (induct k n rule: bools-of-nat.induct)
apply (case-tac n = 0)
apply (case-tac k = 0)
apply simp
apply simp
apply (subst bools-of-nat.simps)
apply (simp add: nat-of-bool-mod-eq [simplified] del: bools-of-nat.simps)
done

declare bools-of-nat.simps [simp del]

lemma eval-dioph-replicate-0: eval-dioph ks (replicate n 0) = 0
apply (induct n arbitrary: ks)
apply simp
apply (case-tac ks)
apply simp-all
done

lemma dioph-dfs-bij:
  (fst (dioph-dfs n ks l) ! int-encode i = Some k ∧ dioph-is-node ks l i =
   (k < length (snd (dioph-dfs n ks l)) ∧ (snd (dioph-dfs n ks l) ! k = i)))
proof –
  let ?dfs = gen-dfs (dioph-succs n ks) dioph-ins dioph-memb (dioph-empt ks l) [l]
  have list-all (dioph-is-node ks l) [l]
    by (simp add: dioph-is-node-def)
  with dioph-dfs.empty-invariant [of ks l]
  have (fst ?dfs ! int-encode i = Some k ∧ dioph-is-node ks l i =
    (k < length (snd ?dfs) ∧ (snd ?dfs ! k = i)))
  proof (induct rule: dioph-dfs.dfs-invariant)
    case base
    show ?case
    by (auto simp add: dioph-empt-def dioph-is-node-def int-encode-bound)
  next
    case (step S y)
    then show ?case
    by (cases y = i)
    (auto simp add: dioph-ins-def dioph-memb-def dioph-is-node-def split-beta
     dioph-invariant-def
     int-encode-bound nth-append inj-eq [OF inj-int-encode])
  qed
  then show ?thesis by (simp add: dioph-dfs-def)
qed

```

lemma *dioph-dfs-mono*:
assumes z : *dioph-invariant* ks l z
and xs : *list-all* (*dioph-is-node* ks l) xs
and H : $\text{fst } z \ ! \ i = \text{Some } k$
shows $\text{fst } (\text{gen-dfs } (\text{dioph-succs } n \ ks) \ \text{dioph-ins } \ \text{dioph-memb } z \ xs) \ ! \ i = \text{Some } k$
using z xs H
apply (*rule* *dioph-dfs.dfs-invariant*)
apply (*simp add*: *dioph-ins-def* *dioph-memb-def* *split-paired-all*)
apply (*case-tac* $i = \text{int-encode } x$)
apply *simp-all*
done

lemma *dioph-dfs-start*:
 $\text{fst } (\text{dioph-dfs } n \ ks \ l) \ ! \ \text{int-encode } l = \text{Some } 0$
apply (*simp add*: *dioph-dfs-def* *gen-dfs-simps* *dioph-dfs.empt* *dioph-is-node-def*)
apply (*rule* *dioph-dfs-mono* [$of - l$])
apply (*rule* *dioph-dfs.ins-invariant*)
apply (*simp add*: *dioph-is-node-def*)
apply (*rule* *dioph-dfs.empt-invariant*)
apply (*simp add*: *dioph-dfs.empt* *dioph-is-node-def*)
apply (*simp add*: *dioph-dfs.succs-is-node* *dioph-is-node-def*)
apply (*simp add*: *dioph-ins-def* *dioph-empt-def* *int-encode-bound* *dioph-is-node-def*)
done

lemma *eq-dfa-error*: $\neg \text{dfa-accepting } (\text{eq-dfa } n \ ks \ l) \ (\text{dfa-steps } (\text{eq-dfa } n \ ks \ l) \ (\text{length } (\text{snd } (\text{dioph-dfs } n \ ks \ l)))) \ bss$
apply (*induct* bss)
apply (*simp add*: *eq-dfa-def* *split-beta* *dfa-accepting-def* *nth-append*)
apply (*simp add*: *eq-dfa-def* *split-beta* *nth-append* *dfa-trans-def*)
done

lemma *eq-dfa-accepting*:
 $(l, m) \in (\text{succsr } (\text{dioph-succs } n \ ks))^* \implies \text{list-all } (\text{is-alph } n) \ bss \implies$
 $\text{dfa-accepting } (\text{eq-dfa } n \ ks \ l) \ (\text{dfa-steps } (\text{eq-dfa } n \ ks \ l) \ (\text{the } (\text{fst } (\text{dioph-dfs } n \ ks \ l) \ ! \ \text{int-encode } m))) \ bss =$
 $(\text{eval-dioph } ks \ (\text{nats-of-boolss } n \ bss) = m)$
proof (*induct* bss *arbitrary*: m)
case *Nil*
have l : *dioph-is-node* ks l l
by (*simp add*: *dioph-is-node-def*)
with $\langle (l, m) \in (\text{succsr } (\text{dioph-succs } n \ ks))^* \rangle$
have m : *dioph-is-node* ks l m
by (*rule* *dioph-dfs.succsr-is-node*)
with l *Nil*
have *dioph-memb* m (*dioph-dfs* n ks l)
by (*simp add*: *dioph-dfs.dfs-eq-rtrancl* *dioph-dfs-def*)
then obtain k **where** k : $\text{fst } (\text{dioph-dfs } n \ ks \ l) \ ! \ \text{int-encode } m = \text{Some } k$
by (*auto simp add*: *dioph-memb-def*)


```

with  $m$  have  $k < \text{length} (\text{snd} (\text{dioph-dfs } n \text{ } ks \text{ } l)) \wedge (\text{snd} (\text{dioph-dfs } n \text{ } ks \text{ } l) ! k = m)$ 
  by (simp add: dioph-dfs-bij [symmetric])
with  $k$  show ?case
  by (simp add: eval-dioph-replicate-0 dfa-accepting-def eq-dfa-def split-beta nth-append)
next
case (Cons bs bss)
have  $l: \text{dioph-is-node } ks \text{ } l \text{ } l$ 
  by (simp add: dioph-is-node-def)
with  $\langle (l, m) \in (\text{succsr} (\text{dioph-succs } n \text{ } ks))^* \rangle$ 
have  $m: \text{dioph-is-node } ks \text{ } l \text{ } m$ 
  by (rule dioph-dfs.succsr-is-node)
with  $l \text{ } Cons$ 
have  $\text{dioph-memb } m (\text{dioph-dfs } n \text{ } ks \text{ } l)$ 
  by (simp add: dioph-dfs.dfs-eq-rtrancl dioph-dfs-def)
then obtain  $k$  where  $k: \text{fst} (\text{dioph-dfs } n \text{ } ks \text{ } l) ! \text{int-encode } m = \text{Some } k$ 
  by (auto simp add: dioph-memb-def)
with  $m$  have  $k': k < \text{length} (\text{snd} (\text{dioph-dfs } n \text{ } ks \text{ } l)) \wedge (\text{snd} (\text{dioph-dfs } n \text{ } ks \text{ } l) ! k = m)$ 
  by (simp add: dioph-dfs-bij [symmetric])
show ?case
proof (cases eval-dioph ks (map nat-of-bool bs) mod 2 = m mod 2)
  case True
  with  $k' \text{ } Cons$ 
  have  $\text{bdd-lookup} (\text{fst} (\text{eq-dfa } n \text{ } ks \text{ } l) ! k) \text{ } bs = \text{the} (\text{fst} (\text{dioph-dfs } n \text{ } ks \text{ } l) ! \text{int-encode} ((m - \text{eval-dioph } ks (\text{map nat-of-bool } bs)) \text{div } 2))$ 
  by (simp add: eq-dfa-def split-beta nth-append bdd-lookup-make-bdd is-alph-def)
  moreover have  $(l, (m - \text{eval-dioph } ks (\text{map nat-of-bool } bs)) \text{div } 2) \in (\text{succsr} (\text{dioph-succs } n \text{ } ks))^*$ 
    apply (rule rtrancl-into-rtrancl)
    apply (rule Cons)
    apply (simp add: dioph-succs-def succsr-def map-filter-def)
    apply (rule image-eqI [of - - map nat-of-bool bs])
    using Cons
    apply (simp-all add: True nat-of-bool-mk-nat-vecs is-alph-def)
    done
  ultimately show ?thesis using True k k' Cons
  by (subst eval-dioph-div-mod)
    (simp add: nats-of-boolss-div2 nats-of-boolss-mod2 is-alph-def dfa-trans-def [abs-def])
next
case False
with  $k' \text{ } Cons$ 
have  $\text{bdd-lookup} (\text{fst} (\text{eq-dfa } n \text{ } ks \text{ } l) ! k) \text{ } bs = \text{length} (\text{snd} (\text{dioph-dfs } n \text{ } ks \text{ } l))$ 
  by (simp add: eq-dfa-def split-beta nth-append bdd-lookup-make-bdd is-alph-def)
with False k k' Cons show ?thesis
  by (subst eval-dioph-div-mod)
    (simp add: nats-of-boolss-div2 nats-of-boolss-mod2 is-alph-def)

```

dfa-trans-def eq-dfa-error)

qed
qed

lemma *eq-dfa-accepts*:
assumes *bss*: *list-all (is-alf n) bss*
shows *dfa-accepts (eq-dfa n ks l) bss = (eval-dioph ks (nats-of-boolss n bss) = l)*
by (*simp add: accepts-def*)
(*rule eq-dfa-accepting [of l l n ks, OF - bss, simplified dioph-dfs-start, simplified]*)

lemma *bddh-make-bdd*: *bddh n (make-bdd f n xs)*
by (*induct n arbitrary: xs simp-all*)

lemma *bdd-all-make-bdd*: *bdd-all P (make-bdd f n xs) = (∀ ys ∈ set (mk-nat-vecs n). P (f (xs @ ys)))*
by (*induct n arbitrary: xs (auto simp add: Let-def)*)

lemma *eq-wf-dfa*: *wf-dfa (eq-dfa n ks l) n*
proof –
have $\forall x \in \text{set } (\text{snd } (\text{dioph-dfs } n \text{ ks } l)). \forall ys \in \text{set } (\text{mk-nat-vecs } n).$
 $\text{eval-dioph } ks \text{ ys mod } 2 = x \text{ mod } 2 \longrightarrow$
 $\text{the } (\text{fst } (\text{dioph-dfs } n \text{ ks } l) ! \text{int-encode } ((x - \text{eval-dioph } ks \text{ ys}) \text{ div } 2)) <$
 $\text{Suc } (\text{length } (\text{snd } (\text{dioph-dfs } n \text{ ks } l)))$
proof (*intro ballI impI*)
fix *x ys*
assume *x*: $x \in \text{set } (\text{snd } (\text{dioph-dfs } n \text{ ks } l))$
and *ys*: $ys \in \text{set } (\text{mk-nat-vecs } n)$
and *ys'*: $\text{eval-dioph } ks \text{ ys mod } 2 = x \text{ mod } 2$
from *x* **obtain** *k* **where** *k*: $\text{fst } (\text{dioph-dfs } n \text{ ks } l) ! \text{int-encode } x = \text{Some } k$
and *k'*: *dioph-is-node ks l x*
by (*auto simp add: in-set-conv-nth dioph-dfs-bij [symmetric]*)
from *k* **have** *dioph-memb x (dioph-dfs n ks l)*
by (*simp add: dioph-memb-def split-beta*)
moreover **have** *ll*: *dioph-is-node ks l l*
by (*simp add: dioph-is-node-def*)
ultimately **have** $(l, x) \in (\text{succsr } (\text{dioph-succs } n \text{ ks}))^*$ **using** *k'*
by (*simp add: dioph-dfs.dfs-eq-rtrancl dioph-dfs-def*)
then **have** $(l, (x - \text{eval-dioph } ks \text{ ys}) \text{ div } 2) \in (\text{succsr } (\text{dioph-succs } n \text{ ks}))^*$
apply (*rule rtrancl-into-rtrancl*)
apply (*simp add: succsr-def dioph-succs-def map-filter-def*)
apply (*rule image-eqI [of - - ys]*)
apply (*simp-all add: ys ys'*)
done
moreover **from** *dioph-dfs.succs-is-node [OF k', of n] ys ys'*
have *x'*: *dioph-is-node ks l ((x - eval-dioph ks ys) div 2)*
by (*auto simp add: dioph-succs-def map-filter-def list-all-iff*)
ultimately **have** *dioph-memb ((x - eval-dioph ks ys) div 2) (dioph-dfs n ks l)*
by (*simp add: dioph-dfs.dfs-eq-rtrancl dioph-dfs-def ll*)
then **obtain** *k'* **where** *k'*: $\text{fst } (\text{dioph-dfs } n \text{ ks } l) !$

```

    int-encode ((x - eval-dioph ks ys) div 2) = Some k'
  by (auto simp add: dioph-memb-def)
with x' have k' < length (snd (dioph-dfs n ks l)) ∧
  snd (dioph-dfs n ks l) ! k' = ((x - eval-dioph ks ys) div 2)
  by (simp add: dioph-dfs-bij [symmetric])
with k'
show the (fst (dioph-dfs n ks l) ! int-encode ((x - eval-dioph ks ys) div 2)) <
  Suc (length (snd (dioph-dfs n ks l)))
  by simp
qed
then show ?thesis
  by (simp add: eq-dfa-def split-beta wf-dfa-def dfa-is-node-def list-all-iff
    bddh-make-bdd bdd-all-make-bdd)
qed

```

5.8 Diophantine Inequations

definition

```

dioph-ineq-succs n ks m = map (λxs.
  (m - eval-dioph ks xs) div 2) (mk-nat-vecs n)

```

interpretation *dioph-ineq-dfs*: DFS *dioph-ineq-succs n ks dioph-is-node ks l*
dioph-invariant ks l dioph-ins dioph-memb dioph-empt ks l

proof (*standard, goal-cases*)

```

case (1 x y)
then show ?case
  apply (simp add: dioph-memb-def dioph-ins-def split-beta dioph-invariant-def)
  apply (cases x = y)
  apply (simp add: int-encode-bound)
  apply (simp add: inj-eq [OF inj-int-encode])
  done
next
case 2
then show ?case
  by (simp add: dioph-memb-def dioph-empt-def int-encode-bound)
next
case 3
then show ?case
  apply (simp add: dioph-ineq-succs-def map-filter-def list-all-iff dioph-is-node-def)
  apply (rule ballI)
  apply (erule subst [OF mk-nat-vecs-mod-eq])
  apply (drule dioph-rhs-invariant)
  apply assumption
  done
next
case 4
then show ?case
  by (simp add: dioph-invariant-def dioph-empt-def)
next

```

```

case 5
then show ?case
  by (simp add: dioph-invariant-def dioph-ins-def split-beta)
next
case 6
then show ?case
  apply (rule bounded-int-set-is-finite [of - max |l| ( $\sum_{k \leftarrow ks} |k|$ ) + 1])
  apply (rule ballI)
  apply (simp add: dioph-is-node-def)
  done
qed

```

definition

$dioph-ineq-dfs\ n\ ks\ l = gen-dfs\ (dioph-ineq-succs\ n\ ks)\ dioph-ins\ dioph-memb\ (dioph-empt\ ks\ l)\ [l]$

definition

$ineq-dfa\ n\ ks\ l =$
 (let (is, js) = $dioph-ineq-dfs\ n\ ks\ l$
 in
 (map ($\lambda j.$ make-bdd ($\lambda xs.$
 the (is ! int-encode ((j - eval-dioph ks xs) div 2))) n []) js,
 map ($\lambda j.$ 0 ≤ j) js))

lemma *dioph-ineq-dfs-bij*:

(fst (dioph-ineq-dfs n ks l) ! int-encode i = Some k ∧ dioph-is-node ks l i) =
 (k < length (snd (dioph-ineq-dfs n ks l)) ∧ (snd (dioph-ineq-dfs n ks l) ! k = i))

proof –

```

let ?dfs = gen-dfs (dioph-ineq-succs n ks) dioph-ins dioph-memb (dioph-empt ks
l) [l]
have list-all (dioph-is-node ks l) [l]
  by (simp add: dioph-is-node-def)
with dioph-dfs.empty-invariant [of ks l]
have (fst ?dfs ! int-encode i = Some k ∧ dioph-is-node ks l i) =
  (k < length (snd ?dfs) ∧ (snd ?dfs ! k = i))
proof (induct rule: dioph-ineq-dfs.dfs-invariant)
  case base
  show ?case
  by (auto simp add: dioph-empt-def dioph-is-node-def int-encode-bound)
next
case (step S y)
then show ?case
  by (cases y = i)
  (auto simp add: dioph-ins-def dioph-memb-def dioph-is-node-def split-beta
dioph-invariant-def
int-encode-bound nth-append inj-eq [OF inj-int-encode])
qed
then show ?thesis by (simp add: dioph-ineq-dfs-def)
qed

```

```

lemma dioph-ineq-dfs-mono:
  assumes z: dioph-invariant ks l z
  and xs: list-all (dioph-is-node ks l) xs
  and H: fst z ! i = Some k
  shows fst (gen-dfs (dioph-ineq-sucss n ks) dioph-ins dioph-memb z xs) ! i = Some
k
  using z xs H
  apply (rule dioph-ineq-dfs.dfs-invariant)
  apply (simp add: dioph-ins-def dioph-memb-def split-paired-all)
  apply (case-tac i = int-encode x)
  apply simp-all
  done

lemma dioph-ineq-dfs-start:
  fst (dioph-ineq-dfs n ks l) ! int-encode l = Some 0
  apply (simp add: dioph-ineq-dfs-def gen-dfs-simps dioph-ineq-dfs.empty dioph-is-node-def)
  apply (rule dioph-ineq-dfs-mono [of - l])
  apply (rule dioph-ineq-dfs.ins-invariant)
  apply (simp add: dioph-is-node-def)
  apply (rule dioph-ineq-dfs.empty-invariant)
  apply (simp add: dioph-ineq-dfs.empty dioph-is-node-def)
  apply (simp add: dioph-ineq-dfs.sucss-is-node dioph-is-node-def)
  apply (simp add: dioph-ins-def dioph-empty-def int-encode-bound dioph-is-node-def)
  done

lemma ineq-dfa-accepting:
   $(l, m) \in (\text{succsr } (\text{dioph-ineq-sucss } n \text{ ks}))^* \implies \text{list-all } (\text{is-alph } n) \text{ bss} \implies$ 
 $\text{dfa-accepting } (\text{ineq-dfa } n \text{ ks } l) (\text{dfa-steps } (\text{ineq-dfa } n \text{ ks } l) (\text{the } (\text{fst } (\text{dioph-ineq-dfs}$ 
 $n \text{ ks } l) ! \text{int-encode } m)) \text{ bss}) =$ 
 $(\text{eval-dioph } ks (\text{nats-of-boolss } n \text{ bss}) \leq m)$ 
  proof (induct bss arbitrary: m)
  case Nil
  have l: dioph-is-node ks l l
    by (simp add: dioph-is-node-def)
  with  $\langle (l, m) \in (\text{succsr } (\text{dioph-ineq-sucss } n \text{ ks}))^* \rangle$ 
  have m: dioph-is-node ks l m
    by (rule dioph-ineq-dfs.sucssr-is-node)
  with l Nil
  have dioph-memb m (dioph-ineq-dfs n ks l)
    by (simp add: dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def)
  then obtain k where k: fst (dioph-ineq-dfs n ks l) ! int-encode m = Some k
    by (auto simp add: dioph-memb-def)
  with m have  $k < \text{length } (\text{snd } (\text{dioph-ineq-dfs } n \text{ ks } l)) \wedge (\text{snd } (\text{dioph-ineq-dfs } n$ 
 $\text{ks } l) ! k = m)$ 
    by (simp add: dioph-ineq-dfs-bij [symmetric])
  with k show ?case
    by (simp add: eval-dioph-replicate-0 dfa-accepting-def ineq-dfa-def split-beta
nth-append)

```

next
case (*Cons bs bss*)
have *l*: *dioph-is-node ks l l*
by (*simp add: dioph-is-node-def*)
with $\langle (l, m) \in (\text{succsr } (\text{dioph-ineq-succs } n \text{ ks}))^* \rangle$
have *m*: *dioph-is-node ks l m*
by (*rule dioph-ineq-dfs.succsr-is-node*)
with *l Cons*
have *dioph-memb m (dioph-ineq-dfs n ks l)*
by (*simp add: dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def*)
then obtain *k* **where** *k*: *fst (dioph-ineq-dfs n ks l) ! int-encode m = Some k*
by (*auto simp add: dioph-memb-def*)
with *m* **have** *k'*: $k < \text{length } (\text{snd } (\text{dioph-ineq-dfs } n \text{ ks } l)) \wedge (\text{snd } (\text{dioph-ineq-dfs } n \text{ ks } l) ! k = m)$
by (*simp add: dioph-ineq-dfs-bij [symmetric]*)
moreover with *Cons* **have** *bdd-lookup (fst (ineq-dfa n ks l) ! k) bs =*
the (fst (dioph-ineq-dfs n ks l) ! int-encode ((m - eval-dioph ks (map nat-of-bool bs)) div 2))
by (*simp add: ineq-dfa-def split-beta nth-append bdd-lookup-make-bdd is-alph-def*)
moreover have $(l, (m - \text{eval-dioph } ks \text{ (map nat-of-bool bs)) div 2) \in (\text{succsr } (\text{dioph-ineq-succs } n \text{ ks}))^*$
apply (*rule rtrancl-into-rtrancl*)
apply (*rule Cons*)
apply (*simp add: dioph-ineq-succs-def succsr-def*)
apply (*rule image-eqI [of - - map nat-of-bool bs]*)
using *Cons*
apply (*simp-all add: nat-of-bool-mk-nat-vecs is-alph-def*)
done
ultimately show *?case* **using** *k Cons*
by (*subst eval-dioph-ineq-div-mod*)
(simp add: nats-of-boolss-div2 nats-of-boolss-mod2 is-alph-def dfa-trans-def [abs-def])
qed

lemma *ineq-dfa-accepts*:
assumes *bss*: *list-all (is-alph n) bss*
shows *dfa-accepts (ineq-dfa n ks l) bss = (eval-dioph ks (nats-of-boolss n bss) \leq l)*
by (*simp add: accepts-def*)
(rule ineq-dfa-accepting [of l l n ks, OF - bss, simplified dioph-ineq-dfs-start, simplified])

lemma *ineq-wf-dfa*: *wf-dfa (ineq-dfa n ks l) n*

proof –

have $\forall x \in \text{set } (\text{snd } (\text{dioph-ineq-dfs } n \text{ ks } l)). \forall ys \in \text{set } (\text{mk-nat-vecs } n).$
the (fst (dioph-ineq-dfs n ks l) ! int-encode ((x - eval-dioph ks ys) div 2)) <
length (snd (dioph-ineq-dfs n ks l))

proof (*intro ballI impI*)

fix *x ys*

```

assume  $x: x \in \text{set } (\text{snd } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l))$ 
and  $ys: ys \in \text{set } (\text{mk-nat-vecs } n)$ 
from  $x$  obtain  $k$  where  $k: \text{fst } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) ! \text{int-encode } x = \text{Some } k$ 
  and  $k': \text{dioph-is-node } ks \text{ } l \text{ } x$ 
  by (auto simp add: in-set-conv-nth dioph-ineq-dfs-bij [symmetric])
from  $k$  have  $\text{dioph-memb } x \text{ } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l)$ 
  by (simp add: dioph-memb-def split-beta)
moreover have  $ll: \text{dioph-is-node } ks \text{ } l \text{ } l$ 
  by (simp add: dioph-is-node-def)
ultimately have  $(l, x) \in (\text{succsr } (\text{dioph-ineq-succs } n \text{ } ks))^* \text{ using } k'$ 
  by (simp add: dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def)
then have  $(l, (x - \text{eval-dioph } ks \text{ } ys) \text{ div } 2) \in (\text{succsr } (\text{dioph-ineq-succs } n \text{ } ks))^*$ 
  apply (rule rtrancl-into-rtrancl)
  apply (simp add: succsr-def dioph-ineq-succs-def)
  apply (rule image-eqI [of - - ys])
  apply (simp-all add: ys)
  done
moreover from  $\text{dioph-ineq-dfs.succs-is-node } [OF \text{ } k', \text{ of } n] \text{ } ys$ 
have  $x': \text{dioph-is-node } ks \text{ } l \text{ } ((x - \text{eval-dioph } ks \text{ } ys) \text{ div } 2)$ 
  by (simp add: dioph-ineq-succs-def list-all-iff)
ultimately have  $\text{dioph-memb } ((x - \text{eval-dioph } ks \text{ } ys) \text{ div } 2) \text{ } (\text{dioph-ineq-dfs } n$ 
 $ks \text{ } l)$ 
  by (simp add: dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def ll)
then obtain  $k'$  where  $k': \text{fst } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) !$ 
 $\text{int-encode } ((x - \text{eval-dioph } ks \text{ } ys) \text{ div } 2) = \text{Some } k'$ 
  by (auto simp add: dioph-memb-def)
with  $x'$  have  $k' < \text{length } (\text{snd } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l)) \wedge$ 
 $\text{snd } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) ! k' = ((x - \text{eval-dioph } ks \text{ } ys) \text{ div } 2)$ 
  by (simp add: dioph-ineq-dfs-bij [symmetric])
with  $k'$ 
show  $\text{the } (\text{fst } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) ! \text{int-encode } ((x - \text{eval-dioph } ks \text{ } ys) \text{ div}$ 
 $2)) <$ 
 $\text{length } (\text{snd } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l))$ 
  by simp
qed
moreover have  $\text{fst } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) ! \text{int-encode } l = \text{Some } 0 \wedge \text{dioph-is-node}$ 
 $ks \text{ } l \text{ } l$ 
  by (simp add: dioph-ineq-dfs-start dioph-is-node-def)
then have  $\text{snd } (\text{dioph-ineq-dfs } n \text{ } ks \text{ } l) \neq []$ 
  by (simp add: dioph-ineq-dfs-bij)
ultimately show ?thesis
  by (simp add: ineq-dfa-def split-beta wf-dfa-def dfa-is-node-def list-all-iff
 $\text{bddh-make-bdd bdd-all-make-bdd}$ )
qed

```

6 Presburger Arithmetic

```

datatype  $pf =$ 
   $\text{Eq int list int}$ 

```

| *Le int list int*
 | *And pf pf*
 | *Or pf pf*
 | *Imp pf pf*
 | *Forall pf*
 | *Exist pf*
 | *Neg pf*

type-synonym *passign = nat list*

primrec *eval-pf* :: *pf* \Rightarrow *passign* \Rightarrow *bool*

where

eval-pf (*Eq ks l*) *xs* = (*eval-dioph ks xs* = *l*)
 | *eval-pf* (*Le ks l*) *xs* = (*eval-dioph ks xs* \leq *l*)
 | *eval-pf* (*And p q*) *xs* = (*eval-pf p xs* \wedge *eval-pf q xs*)
 | *eval-pf* (*Or p q*) *xs* = (*eval-pf p xs* \vee *eval-pf q xs*)
 | *eval-pf* (*Imp p q*) *xs* = (*eval-pf p xs* \longrightarrow *eval-pf q xs*)
 | *eval-pf* (*Forall p*) *xs* = ($\forall x.$ *eval-pf p* (*x # xs*))
 | *eval-pf* (*Exist p*) *xs* = ($\exists x.$ *eval-pf p* (*x # xs*))
 | *eval-pf* (*Neg p*) *xs* = (\neg *eval-pf p xs*)

function *dfa-of-pf* :: *nat* \Rightarrow *pf* \Rightarrow *dfa*

where

Eq: *dfa-of-pf n* (*Eq ks l*) = *eq-dfa n ks l*
 | *Le*: *dfa-of-pf n* (*Le ks l*) = *ineq-dfa n ks l*
 | *And*: *dfa-of-pf n* (*And p q*) = *and-dfa* (*dfa-of-pf n p*) (*dfa-of-pf n q*)
 | *Or*: *dfa-of-pf n* (*Or p q*) = *or-dfa* (*dfa-of-pf n p*) (*dfa-of-pf n q*)
 | *Imp*: *dfa-of-pf n* (*Imp p q*) = *imp-dfa* (*dfa-of-pf n p*) (*dfa-of-pf n q*)
 | *Exist*: *dfa-of-pf n* (*Exist p*) = *rquot* (*det-nfa* (*quantify-nfa* 0 (*nfa-of-dfa* (*dfa-of-pf* (*Suc n*) *p*)))) *n*
 | *Forall*: *dfa-of-pf n* (*Forall p*) = *dfa-of-pf n* (*Neg* (*Exist* (*Neg p*)))
 | *Neg*: *dfa-of-pf n* (*Neg p*) = *negate-dfa* (*dfa-of-pf n p*)
by *pat-completeness auto*

Auxiliary measure function for termination proof

primrec *count-forall* :: *pf* \Rightarrow *nat*

where

count-forall (*Eq ks l*) = 0
 | *count-forall* (*Le ks l*) = 0
 | *count-forall* (*And p q*) = *count-forall p* + *count-forall q*
 | *count-forall* (*Or p q*) = *count-forall p* + *count-forall q*
 | *count-forall* (*Imp p q*) = *count-forall p* + *count-forall q*
 | *count-forall* (*Exist p*) = *count-forall p*
 | *count-forall* (*Forall p*) = 1 + *count-forall p*
 | *count-forall* (*Neg p*) = *count-forall p*

termination *dfa-of-pf*

by (*relation measures* [$\lambda(n, pf).$ *count-forall pf*, $\lambda(n, pf).$ *size pf*]) *auto*


```

lemmas dfa-of-pf-induct =
  dfa-of-pf.induct [case-names Eq Le And Or Imp Exist Forall Neg]

lemma dfa-of-pf-well-formed: wf-dfa (dfa-of-pf n p) n
proof (induct n p rule: dfa-of-pf-induct)
  case (Eq n ks l)
    show ?case by (simp add: eq-wf-dfa)
next
  case (Le n ks l)
    show ?case by (simp add: ineq-wf-dfa)
next
  case (And n p q)
    then show ?case by (simp add: and-wf-dfa)
next
  case (Or n p q)
    then show ?case by (simp add: or-wf-dfa)
next
  case (Imp n p q)
    then show ?case by (simp add: imp-wf-dfa)
next
  case (Neg n p)
    then show ?case by (simp add: negate-wf-dfa)
next
  case (Exist n p)
    then show ?case
      by (simp add: rquot-well-formed-aut det-wf-nfa quantify-nfa-well-formed-aut
dfa2wf-nfa)
next
  case (Forall n p)
    then show ?case by simp
qed

lemma dfa-of-pf-correctness:
  list-all (is-alph n) bss  $\implies$ 
  dfa-accepts (dfa-of-pf n p) bss = eval-pf p (nats-of-boolss n bss)
proof (induct n p arbitrary: bss rule: dfa-of-pf-induct)
  case (Eq n ks l)
    then show ?case by (simp add: eq-dfa-accepts)
next
  case (Le n ks l)
    then show ?case by (simp add: ineq-dfa-accepts)
next
  case (And n p q)
    then show ?case by (simp add: and-dfa-accepts [of - n] dfa-of-pf-well-formed)
next
  case (Or n p q)
    then show ?case by (simp add: or-dfa-accepts [of - n] dfa-of-pf-well-formed)
next
  case (Imp n p q)

```

```

then show ?case by (simp add: imp-dfa-accepts [of - n] dfa-of-pf-well-formed)
next
  case (Neg n p)
  then show ?case by (simp add: dfa-accepts-negate [of - n] dfa-of-pf-well-formed)
next
  case (Exist n p)
  have ( $\exists k$  bs. eval-pf p (nat-of-bools bs # nats-of-boolss n bss)  $\wedge$  length bs = length
bss + k) =
  ( $\exists x$ . eval-pf p (x # nats-of-boolss n bss))
  apply (rule iffI)
  apply (erule exE conjE)+
  apply (erule exI)
  apply (erule exE)
  apply (rule-tac x=length (bools-of-nat (length bss) x) - length bss in exI)
  apply (rule-tac x=bools-of-nat (length bss) x in exI)
  apply (simp add: bools-of-nat-inverse bools-of-nat-length)
  done
with Exist show ?case by (simp add:
  rquot-accepts det-wf-nfa quantify-nfa-well-formed-aut dfa2wf-nfa
  det-nfa-accepts [of - n] zeros-is-alpha nfa-accepts-quantify-nfa [of - n]
  nfa-of-dfa-accepts [of - Suc n] insertll-len2 nats-of-boolss-insertll
  zeros-len nats-of-boolss-append nats-of-boolss-zeros zip-replicate-mapr
  nats-of-boolss-length o-def insertl-0-eq
  dfa-of-pf-well-formed cong: rev-conj-cong)
next
  case (Forall n p)
  then show ?case by simp
qed

```

The same with minimization after quantification.

```

function dfa-of-pf' :: nat  $\Rightarrow$  pf  $\Rightarrow$  dfa
where
  dfa-of-pf' n (Eq ks l) = eq-dfa n ks l
| dfa-of-pf' n (Le ks l) = ineq-dfa n ks l
| dfa-of-pf' n (And p q) = and-dfa (dfa-of-pf' n p) (dfa-of-pf' n q)
| dfa-of-pf' n (Or p q) = or-dfa (dfa-of-pf' n p) (dfa-of-pf' n q)
| dfa-of-pf' n (Imp p q) = imp-dfa (dfa-of-pf' n p) (dfa-of-pf' n q)
| dfa-of-pf' n (Exist p) = min-dfa (rquot (det-nfa (quantify-nfa 0 (nfa-of-dfa (dfa-of-pf'
(Suc n) p)))) n)
| dfa-of-pf' n (Forall p) = dfa-of-pf' n (Neg (Exist (Neg p)))
| dfa-of-pf' n (Neg p) = negate-dfa (dfa-of-pf' n p)
  by pat-completeness auto

```

```

termination dfa-of-pf'
  by (relation measures [ $\lambda(n, pf)$ . count-forall pf,  $\lambda(n, pf)$ . size pf]) auto

```

```

lemmas dfa-of-pf'-induct =
  dfa-of-pf'.induct [case-names Eq Le And Or Imp Exist Forall Neg]

```

```

lemma dfa-of-pf'-well-formed: wf-dfa (dfa-of-pf' n p) n
proof (induct n p rule: dfa-of-pf'-induct)
  case (Eq n ks l)
    show ?case by (simp add: eq-wf-dfa)
next
  case (Le n ks l)
    show ?case by (simp add: ineq-wf-dfa)
next
  case (And n p q)
    then show ?case by (simp add: and-wf-dfa)
next
  case (Or n p q)
    then show ?case by (simp add: or-wf-dfa)
next
  case (Imp n p q)
    then show ?case by (simp add: imp-wf-dfa)
next
  case (Neg n p)
    then show ?case by (simp add: negate-wf-dfa)
next
  case (Exist n p)
    then show ?case
      by (simp add: rquot-well-formed-aut det-wf-nfa quantify-nfa-well-formed-aut
dfa2wf-nfa min-dfa-wf)
next
  case (Forall n p)
    then show ?case by simp
qed

```

```

lemma dfa-of-pf'-correctness:
  list-all (is-alph n) bss  $\implies$ 
  dfa-accepts (dfa-of-pf' n p) bss = eval-pf p (nats-of-boolss n bss)
proof (induct n p arbitrary: bss rule: dfa-of-pf'-induct)
  case (Eq n ks l)
    then show ?case by (simp add: eq-dfa-accepts)
next
  case (Le n ks l)
    then show ?case by (simp add: ineq-dfa-accepts)
next
  case (And n p q)
    then show ?case by (simp add: and-dfa-accepts [of - n] dfa-of-pf'-well-formed)
next
  case (Or n p q)
    then show ?case by (simp add: or-dfa-accepts [of - n] dfa-of-pf'-well-formed)
next
  case (Imp n p q)
    then show ?case by (simp add: imp-dfa-accepts [of - n] dfa-of-pf'-well-formed)
next
  case (Neg n p)

```

```

then show ?case by (simp add: dfa-accepts-negate [of - n] dfa-of-pf'-well-formed)
next
  case (Exist n p)
  have ( $\exists k$  bs. eval-pf p (nat-of-bools bs # nats-of-boolss n bss)  $\wedge$  length bs = length
bss + k) =
  ( $\exists x$ . eval-pf p (x # nats-of-boolss n bss))
  apply (rule iffI)
  apply (erule exE conjE)+
  apply (erule exI)
  apply (erule exE)
  apply (rule-tac x=length (bools-of-nat (length bss) x) - length bss in exI)
  apply (rule-tac x=bools-of-nat (length bss) x in exI)
  apply (simp add: bools-of-nat-inverse bools-of-nat-length)
  done
with Exist show ?case by (simp add:
  rquot-accepts det-wf-nfa quantify-nfa-well-formed-aut dfa2wf-nfa
  det-nfa-accepts [of - n] zeros-is-alpha nfa-accepts-quantify-nfa [of - n]
  nfa-of-dfa-accepts [of - Suc n] insertll-len2 nats-of-boolss-insertll
  zeros-len nats-of-boolss-append nats-of-boolss-zeros zip-replicate-mapr
  nats-of-boolss-length o-def insertl-0-eq
  dfa-of-pf'-well-formed min-dfa-accept [of - n] min-dfa-wf rquot-well-formed-aut
  cong: rev-conj-cong)
next
  case (Forall n p)
  then show ?case by simp
qed

```

References

- [1] S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 147–163. Springer-Verlag, 2009.
- [2] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.
- [3] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, CSL '97, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, 1998.