Formalizing the Logic-Automaton Connection

Markus Reiter Stefan Berghofer

September 30, 2020

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

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* ⟹ *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* = (∃ *q*. *reach tr p bs q* ∧ *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 ⋀*k x y*. ⟦*reach tr p x k*; *P x k*⟧ ⟹ *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*: ⟦*reach tr p a r*; *reach tr r b q*⟧ ⟹ *reach tr p* (*a* @ *b*) *q*
by (*simp add*: *reach-def*)

lemma *reach-inj*: ⟦*reach tr p a q*; *reach tr p a q'*⟧ ⟹ *q* = *q'*
by (*simp add*: *reach-def*)

definition
accepts tr P s as = *P* (*foldl tr s as*)

locale *Automaton* =
fixes *trans* :: *'a* ⇒ *'b* ⇒ *'a*
and *is-node* :: *'a* ⇒ *bool*
and *is-alpha* :: *'b* ⇒ *bool*
assumes *trans-is-node*: ⋀*q a*. ⟦*is-node q*; *is-alpha a*⟧ ⟹ *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*: ⟦*reach trans p w q*; *is-node p*; *list-all is-alpha w*⟧ ⟹ *is-node q*
by (*simp add*: *steps-is-node reach-def*)

end

2 BDDs

definition
is-alph :: *nat* ⇒ *bool list* ⇒ *bool* **where**
is-alph n = (λ*w*. *length w* = *n*)

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

primrec *bddh* :: *nat* ⇒ *'a bdd* ⇒ *bool*
where
bddh n (*Leaf x*) = *True*
| *bddh n* (*Branch l r*) = (*case n of* 0 ⇒ *False* | *Suc m* ⇒ *bddh m l* ∧ *bddh m r*)

lemma *bddh-ge*:
assumes *m* ≥ *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** ∃ *w*. *m* = *Suc w* ∧ *n* ≤ *w* **by** (*cases m*) *simp+*
then obtain *w* **where** *W*: *m* = *Suc w* ∧ *n* ≤ *w* **..**
with *Branch V* **have** *v* ≤ *w* ∧ *bddh v l* ∧ *bddh v r* **by** *simp*
with *Branch* **have** *bddh w l* ∧ *bddh w r* **by** *blast*
with *W* **show** *?thesis* **by** *simp*
qed
qed *simp*

abbreviation *bdd-all* ≡ *pred-bdd*

fun *bdd-lookup* :: *'a bdd* ⇒ *bool list* ⇒ *'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*: ⟦*bddh* (*length ws*) *bdd*; *bdd-all P bdd*⟧ ⟹ *P* (*bdd-lookup bdd ws*)
by (*induct bdd ws rule*: *bdd-lookup.induct*) *simp+*

lemma *bdd-all-bdd-lookup-iff*: *bddh n bdd* ⟹ *bdd-all P bdd* = (∀ *ws*. *length ws* = *n* ⟶ *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: ⋀ws. length ws = n ⟹ P (bdd-lookup (Branch l r) ws)
by simp
    have ⋀ws. length ws = k ⟹ P (bdd-lookup l ws) ∧ P (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 (bdd-lookup (Branch l r) (False#ws)) by (rule R)
      from H k have length (True#ws) = n by simp
      hence P (bdd-lookup (Branch l r) (True#ws)) by (rule R)
      with 1 show P (bdd-lookup l ws) ∧ P (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 ⋀a. P a ⟹ 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 *bdd-lookup* (*bdd-map f bdd*) *ws* = *f* (*bdd-lookup bdd ws*)
using *assms* **by** (*induct bdd ws rule*: *bdd-lookup.induct*) (*auto simp add*: *bddh-bdd-map*)+

fun *bdd-binop* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a bdd* ⇒ '*b bdd* ⇒ '*c bdd*
where
bdd-binop f (*Leaf x*) (*Leaf y*) = *Leaf* (*f x y*)
| *bdd-binop f* (*Branch l r*) (*Leaf y*) = *Branch* (*bdd-binop f l* (*Leaf y*)) (*bdd-binop f r* (*Leaf y*))
| *bdd-binop f* (*Leaf x*) (*Branch l r*) = *Branch* (*bdd-binop f* (*Leaf x*) *l*) (*bdd-binop f* (*Leaf x*) *r*)
| *bdd-binop f* (*Branch* l_1 r_1) (*Branch* l_2 r_2) = *Branch* (*bdd-binop f* l_1 l_2) (*bdd-binop f* r_1 r_2)

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

lemma *bdd-lookup-binop*: ⟦*bddh* (*length bs*) *l*; *bddh* (*length bs*) *r*⟧ ⟹
bdd-lookup (*bdd-binop f l r*) *bs* = *f* (*bdd-lookup l bs*) (*bdd-lookup r 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 ⋀*a b*. ⟦*P a*; *Q b*⟧ ⟹ *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*]:
List.insert x (*List.insert x xs*) = *List.insert x xs*
by *simp*

primrec *add-leaves* :: '*a bdd* ⇒ '*a list* ⇒ '*a list*
where
add-leaves (*Leaf x*) *xs* = *List.insert x xs*
| *add-leaves* (*Branch b c*) *xs* = *add-leaves c* (*add-leaves b xs*)

lemma *add-leaves-bdd-lookup*:
bddh n b ⟹ (*x* ∈ *set* (*add-leaves b xs*)) = ((∃ *bs*. *x* = *bdd-lookup b bs* ∧ *is-alph n bs*) ∨ *x* ∈ *set 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-all P (*add-leaves tr xs*) ⟷ *bdd-all P tr* ∧ *list-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 ⊆ *set ys* ⟹ *set* (*add-leaves tr xs*) ⊆ *set* (*add-leaves tr ys*)
by (*induct tr arbitrary*: *xs ys*) *auto*

lemma *add-leaves-binop-subset*:
set (*add-leaves* (*bdd-binop f b b'*) [*f x y*. *x* ← *xs*, *y* ← *ys*]) ⊆
(⋃*x*∈*set* (*add-leaves b xs*). ⋃*y*∈*set* (*add-leaves b' ys*). {*f x y*}) (**is** *?A* ⊆ *?B*)
proof −
have *?A* ⊆ (⋃*x*∈*set* (*add-leaves b xs*). *f x* ' *set* (*add-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 ys*1=[*f x y*. *x* ← *add-leaves* l_1 *xs*, *y* ← *add-leaves* l_2 *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 ($\bigcup x \in set$ (*add-leaves b xs*). *f x* ‘ *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 = (λ*q*. *q* < *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*: *dfa-is-node A q* **and** *A*: *is-alph n a*
hence *QL*: *q* < *length* (*fst A*) **by** (*simp add*: *dfa-is-node-def*)
with *well-formed A* **have** *H*: *bddh* (*length a*) (*fst A* ! *q*) **by** (*simp add*: *wf-dfa-def list-all-iff is-alph-def*)
from *QL well-formed* **have** *bdd-all* (*dfa-is-node A*) (*fst A* ! *q*) **by** (*simp add*: *wf-dfa-def list-all-iff*)
with *H* **show** *dfa-is-node A* (*dfa-trans A q a*) **by** (*simp add*: *dfa-trans-def bdd-all-bdd-lookup*)
qed

context *aut-dfa* **begin**
lemmas *trans-is-node* = *trans-is-node*
lemmas *steps-is-node* = *steps-is-node*
lemmas *reach-is-node* = *reach-is-node*
end

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

abbreviation *dfa-steps A* ≡ *foldl* (*dfa-trans A*)
abbreviation *dfa-accepts A* ≡ *accepts* (*dfa-trans A*) (*dfa-accepting A*) 0
abbreviation *dfa-reach A* ≡ *reach* (*dfa-trans A*)

lemma *dfa-startnode-is-node*: *wf-dfa A n* ⟹ *dfa-is-node A* 0
by (*simp add*: *dfa-is-node-def wf-dfa-def*)

3.1 Minimization

primrec *make-tr* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *nat* ⇒ *'a list*
where
make-tr f 0 *i* = []
| *make-tr f* (*Suc n*) *i* = *f i* # *make-tr f n* (*Suc i*)

primrec *fold-map-idx* :: (*nat* ⇒ *'c* ⇒ *'a* ⇒ *'c* × *'b*) ⇒ *nat* ⇒ *'c* ⇒ *'a list* ⇒ *'c* × *'b list*
where
fold-map-idx f i y [] = (*y*, [])
| *fold-map-idx f i y* (*x* # *xs*) =
(*let* (*y'*, *x'*) = *f i y x in*
let (*y''*, *xs'*) = *fold-map-idx f* (*Suc i*) *y' xs in* (*y''*, *x'* # *xs'*))

definition *init-tr* :: *dfa* ⇒ *bool list list* **where**
init-tr = (λ(*bd*,*as*). *make-tr* (λ*i*. *make-tr* (λ*j*. *as* ! *i* ≠ *as* ! *j*) *i* 0) (*length bd* − 1) 1)

definition *tr-lookup* :: *bool list list* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**

tr-lookup = (λ*T i j*. (*if i* = *j then False else if i* > *j then T* ! (*i* − 1) ! *j else T* ! (*j* − 1) ! *i*))

fun *check-eq* :: *nat bdd* ⇒ *nat bdd* ⇒ *bool list list* ⇒ *bool* **where**
check-eq (*Leaf i*) (*Leaf j*) *T* = (¬ *tr-lookup T i j*) |
check-eq (*Branch l r*) (*Leaf i*) *T* = (*check-eq l* (*Leaf i*) *T* ∧ *check-eq r* (*Leaf i*) *T*) |
check-eq (*Leaf i*) (*Branch l r*) *T* = (*check-eq* (*Leaf i*) *l T* ∧ *check-eq* (*Leaf i*) *r T*) |
check-eq (*Branch l*1 *r*1) (*Branch l*2 *r*2) *T* = (*check-eq l*1 *l*2 *T* ∧ *check-eq r*1 *r*2 *T*)

definition *iter* :: *dfa* ⇒ *bool list list* ⇒ *bool* × *bool list list* **where**
iter = (λ(*bd*,*as*) *T*. *fold-map-idx* (λ*i*. *fold-map-idx* (λ*j c b*.
let b′ = *b* ∨ ¬ *check-eq* (*bd* ! *i*) (*bd* ! *j*) *T*
in (*c* ∨ *b* ≠ *b*′, *b*′)) 0) 1 *False T*)

definition *count-tr* :: *bool list list* ⇒ *nat* **where**
count-tr = *foldl* (*foldl* (λ*y x*. *if x then y else Suc y*)) 0

lemma *fold-map-idx-fst-snd-eq*:
assumes *f*: ⋀*i c x*. *fst* (*f i c x*) = (*c* ∨ *x* ≠ *snd* (*f i c x*))
shows *fst* (*fold-map-idx f i c xs*) = (*c* ∨ *xs* ≠ *snd* (*fold-map-idx f i c xs*))
by (*induct xs arbitrary*: *i c*) (*simp-all add*: *split-beta f*)

lemma *foldl-mono*:
assumes *f*: ⋀*x y y*′. *y* < *y*′ ⟹ *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-map-idx-count*:
assumes *f*: ⋀*i c x y*. *fst* (*f i c x*) = (*c* ∨ *g y* (*snd* (*f i c x*)) < (*g y x*::*nat*))
and *f*′: ⋀*i c x y*. *g y* (*snd* (*f i c x*)) ≤ *g y x*
and *g*: ⋀*x y y*′. *y* < *y*′ ⟹ *g y x* < *g y*′ *x*
shows *fst* (*fold-map-idx f i c xs*) =
(*c* ∨ *foldl g y* (*snd* (*fold-map-idx f i c xs*)) < *foldl g y xs*)
and *foldl g y* (*snd* (*fold-map-idx f i c xs*)) ≤ *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*: $(b,\ T') = iter\ (bd,\ as)\ T$
  and *b*: *b*
  shows *count-tr T'* < *count-tr T*
proof −
  let *?f* = *fold-map-idx* (λ*i*. *fold-map-idx* (λ*j c b*.
    let b' = *b* ∨ ¬ *check-eq* (*bd* ! *i*) (*bd* ! *j*) *T*
    in (*c* ∨ *b* ≠ *b'*, *b'*)) 0) (*Suc* 0) *False T*
  from *eq* [*symmetric*] *b* **have** *fst ?f*
    by (*auto simp add*: *iter-def*)
  also have *fst ?f* = (*False* ∨ *count-tr* (*snd ?f*) < *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* :: *dfa* ⇒ *bool list list* ⇒ *bool list list* **where**
  fixpt M T = (*let* (*b*, *T2*) = *iter M T in if b then fixpt M T2 else T2*)
by *auto*
termination by (*relation measure* (λ(*M*, *T*). *count-tr T*)) (*auto simp*: *iter-count*)

lemma *fixpt-True*[*simp*]: *fst* (*iter M T*) ⟹ *fixpt M T* = *fixpt M* (*snd* (*iter M T*))
  by (*simp add*: *split-beta*)

lemma *fixpt-False*[*simp*]: ¬ (*fst* (*iter M T*)) ⟹ *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*: ⋀*M T*. (*fst* (*iter M T*) ⟹ *P M* (*snd* (*iter M T*))) ⟹ *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* :: *dfa* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  dist-nodes = (λ*M n m p q*. ∃ *w*. *length w* = *n* ∧ *list-all* (*is-alph m*) *w* ∧
    dfa-accepting M (*dfa-steps M p w*) ≠ *dfa-accepting M* (*dfa-steps M q w*))

definition *wf-tr* :: *dfa* ⇒ *bool list list* ⇒ *bool* **where**
  wf-tr = (λ*M T*. *length T* = *length* (*fst M*) − 1 ∧ (∀ *i* < *length T*. *length* (*T* ! *i*) = *i* + 1))

lemma *make-tr-len*: *length* (*make-tr f n i*) = *n*

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

lemma *make-tr-nth*: $j < n \Longrightarrow$ *make-tr f n i* ! *j* = *f* (*i* + *j*)
by (*induct n arbitrary*: *i j*) (*auto simp add*: *nth-Cons'*)

lemma *init-tr-wf*: *wf-tr M* (*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*: *length* (*snd* (*fold-map-idx f i y xs*)) = *length xs*
by (*induct xs arbitrary*: *i y*) (*simp-all add*: *split-beta*)

lemma *fold-map-idx-nth*: *j* < *length xs* $\Longrightarrow$
snd (*fold-map-idx f i y xs*) ! *j* = *snd* (*f* (*i* + *j*) (*fst* (*fold-map-idx f i y* (*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 *tr-lookup* (*init-tr M*) *q p* = *dist-nodes M* 0 *v p q*
proof −
have 1: *dist-nodes M* 0 *v p q* = (*snd M* ! *p* ≠ *snd M* ! *q*) **by** (*simp add*: *dist-nodes-def dfa-accepting-def*)
from *assms* **have** *tr-lookup* (*init-tr M*) *q p* = (*snd M* ! *p* ≠ *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*:
dist-nodes M (*Suc n*) *v p q* = (∃ *bs*. *is-alph v bs* ∧ *dist-nodes M n v* (*dfa-trans M p bs*) (*dfa-trans M q bs*))
proof
assume *dist-nodes M* (*Suc n*) *v p q*
then obtain *w* **where** *W*: *length w* = *Suc n* **and** *L*: *list-all* (*is-alph v*) *w* **and** *A*: *dfa-accepting M* (*dfa-steps M p w*) ≠ *dfa-accepting M* (*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*: *dfa-accepting M* (*dfa-steps M* (*dfa-trans M p b*) *bs*) ≠ *dfa-accepting M* (*dfa-steps M* (*dfa-trans M q b*) *bs*)
unfolding *B* **by** *simp*
with *W B L* **show** ∃ *bs*. *is-alph v bs* ∧ *dist-nodes M n v* (*dfa-trans M p bs*) (*dfa-trans M q bs*) **by** (*auto simp*: *dist-nodes-def*)
next
assume ∃ *bs*. *is-alph v bs* ∧ *dist-nodes M n v* (*dfa-trans M p bs*) (*dfa-trans M q bs*)
then obtain *b bs* **where** *W*: *length bs* = *n* **and** *V*: *is-alph v b* **and** *V'*: *list-all* (*is-alph v*) *bs*
and *A*: *dfa-accepting M* (*dfa-steps M* (*dfa-trans M p b*) *bs*) ≠ *dfa-accepting M* (*dfa-steps M* (*dfa-trans M q b*) *bs*)
unfolding *dist-nodes-def* **by** *blast*

hence *dfa-accepting M* (*dfa-steps M p* (*b # bs*)) ≠ *dfa-accepting M* (*dfa-steps M q* (*b # bs*)) **by** *simp*
moreover from *W* **have** *length* (*b # bs*) = *Suc n* **by** *simp*
moreover from *V V'* **have** *list-all* (*is-alph v*) (*b # bs*) **by** *simp*
ultimately show *dist-nodes M* (*Suc n*) *v p q* **unfolding** *dist-nodes-def* **by** *blast*
qed

lemma *bdd-lookup-append*:
assumes *bddh n B* **and** *length bs* ≥ *n*
shows *bdd-lookup B* (*bs* @ *w*) = *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* = *Suc n'* **by** (*cases n*) *simp+*
with 2 **show** *?case* **by** (*cases b*) *auto*
qed *simp+*

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

lemma *check-eq-dist-nodes*:
assumes ∀ *p q*. *dfa-is-node M q* ∧ *p* < *q* ⟶ *tr-lookup T q p* = (∃ *n* < *m*. *dist-nodes M n v p q*) **and** *m* > 0
and *bdd-all* (*dfa-is-node M*) *l* **and** *bdd-all* (*dfa-is-node M*) *r*
shows (¬ *check-eq l r T*) = (∃ *bs*. *bddh* (*length bs*) *l* ∧ *bddh* (*length bs*) *r* ∧ (∃ *n* < *m*. *dist-nodes M n v* (*bdd-lookup l bs*) (*bdd-lookup r bs*)))
using *assms* **proof** (*induct l r T rule*: *check-eq.induct*)
case (1 *i j T*)
have *i* < *j* ∨ *i* = *j* ∨ *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 *IV*1: (¬ *check-eq l* (*Leaf i*) *T*) = (∃ *bs*. *bddh* (*length bs*) *l* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup l bs*) (*bdd-lookup* (*Leaf i*) *bs*)))
by *simp*
from 2 **have** *IV*2: (¬ *check-eq r* (*Leaf i*) *T*) = (∃ *bs*. *bddh* (*length bs*) *r* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup r bs*) (*bdd-lookup* (*Leaf i*) *bs*))) **by** *simp*
have (¬ *check-eq* (*Branch l r*) (*Leaf i*) *T*) = (¬ *check-eq l* (*Leaf i*) *T* ∨ ¬ *check-eq r* (*Leaf i*) *T*) **by** *simp*
also have ... = (∃ *bs*. *bddh* (*length bs*) (*Branch l r*) ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m* . *dist-nodes M n v* (*bdd-lookup* (*Branch l r*) *bs*) (*bdd-lookup* (*Leaf i*) *bs*)))
(**is** (*?L* ∨ *?R*) = *?E*)

proof
assume *?L* ∨ *?R*
thus *?E* **proof** (*elim disjE*)
assume *?L*
then obtain *bs* **where** *O*: *bddh* (*length bs*) *l* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup l bs*) (*bdd-lookup* (*Leaf i*) *bs*)) **unfolding** *IV*1 **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*) ∧ *bddh* (*length* (*False* # *bs* @ *replicate k False*)) (*Leaf i*) ∧ (∃ *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* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup r bs*) (*bdd-lookup* (*Leaf i*) *bs*)) **unfolding** *IV*2 **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*) ∧ *bddh* (*length* (*True* # *bs* @ *replicate k False*)) (*Leaf i*) ∧ (∃ *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*) ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *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 IV*1 *IV*2 **show** *?L* ∨ *?R* **by** (*cases b*) *auto*
qed
finally show *?case* **by** *simp*
next
case (3 *i l r T*)
hence *IV*1: (¬ *check-eq* (*Leaf i*) *l T*) = (∃ *bs*. *bddh* (*length bs*) *l* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup l bs*))) **by** *simp*
from 3 **have** *IV*2: (¬ *check-eq* (*Leaf i*) *r T*) = (∃ *bs*. *bddh* (*length bs*) *r* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup r bs*))) **by** *simp*
have (¬ *check-eq* (*Leaf i*) (*Branch l r*) *T*) = (¬ *check-eq* (*Leaf i*) *l T* ∨ ¬ *check-eq*

(*Leaf i*) *r T*) **by** *simp*
also have ... = (∃ *bs*. *bddh* (*length bs*) (*Branch l r*) ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m* . *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup* (*Branch l r*) *bs*)))
(**is** (*?L* ∨ *?R*) = *?E*)
proof
assume *?L* ∨ *?R*
thus *?E* **proof** (*elim disjE*)
assume *?L*
then obtain *bs* **where** *O*: *bddh* (*length bs*) *l* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup l bs*)) **unfolding** *IV*1 **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*) ∧ *bddh* (*length* (*False* # *bs* @ *replicate k False*)) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) (*False* # *bs* @ *replicate k False*)) (*bdd-lookup* (*Branch l r*) (*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* ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup r bs*)) **unfolding** *IV*2 **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*) ∧ *bddh* (*length* (*True* # *bs* @ *replicate k False*)) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) (*True* # *bs* @ *replicate k False*)) (*bdd-lookup* (*Branch l r*) (*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*) ∧ *bddh* (*length bs*) (*Leaf i*) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Leaf i*) *bs*) (*bdd-lookup* (*Branch l r*) *bs*)) **by** *blast*
then obtain *b br* **where** *B*: *bs* = *b* # *br* **by** (*cases bs*) *auto*
with *O IV*1 *IV*2 **show** *?L* ∨ *?R* **by** (*cases b*) *auto*
qed
finally show *?case* **by** *simp*
next
case (4 *l*1 *r*1 *l*2 *r*2 *T*)
hence *IV*1: (¬ *check-eq l*1 *l*2 *T*) = (∃ *bs*. *bddh* (*length bs*) *l*1 ∧ *bddh* (*length bs*) *l*2 ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup l*1 *bs*) (*bdd-lookup l*2 *bs*))) **by** *simp*
from 4 **have** *IV*2: (¬ *check-eq r*1 *r*2 *T*) = (∃ *bs*. *bddh* (*length bs*) *r*1 ∧ *bddh*

(*length bs*) *r*2 ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup r*1 *bs*) (*bdd-lookup r*2 *bs*)))
by *simp*
have (¬ *check-eq* (*Branch l*1 *r*1) (*Branch l*2 *r*2) *T*) = (¬ *check-eq l*1 *l*2 *T* ∨ ¬ *check-eq r*1 *r*2 *T*) **by** *simp*
also have ... = (∃ *bs*. *bddh* (*length bs*) (*Branch l*1 *r*1) ∧ *bddh* (*length bs*) (*Branch l*2 *r*2) ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Branch l*1 *r*1) *bs*) (*bdd-lookup* (*Branch l*2 *r*2) *bs*)))
(**is** (*?L* ∨ *?R*) = (∃ *bs*. *?E bs*)) **proof**
assume *?L* ∨ *?R*
thus ∃ *bs*. *?E bs* **proof** (*elim disjE*)
assume *?L*
then obtain *bs* **where** *O*: *bddh* (*length bs*) *l*1 ∧ *bddh* (*length bs*) *l*2 ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup l*1 *bs*) (*bdd-lookup l*2 *bs*)) **unfolding** *IV*1 **by** *blast*
from *bddh-exists* **obtain** *k*1 *k*2 **where** *K*1: *bddh k*1 *r*1 **and** *K*2: *bddh k*2 *r*2 **by** *blast*
with *O* **have** *bddh* (*length bs* + *max k*1 *k*2) *l*1 **and** *bddh* (*length bs* + *max k*1 *k*2) *l*2 **and** *bddh* (*length bs* + *max k*1 *k*2) *r*1 **and** *bddh* (*length bs* + *max k*1 *k*2) *r*2
by (*simp add*: *bddh-ge*[*of length bs length bs* + *max k*1 *k*2] *bddh-ge*[*of k*1 *length bs* + *max k*1 *k*2] *bddh-ge*[*of k*2 *length bs* + *max k*1 *k*2])+
with *O* **have** *bddh* (*length* (*False* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*Branch l*1 *r*1) ∧ *bddh* (*length* (*False* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*Branch l*2 *r*2) ∧
(∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Branch l*1 *r*1) (*False* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*bdd-lookup* (*Branch l*2 *r*2) (*False* # *bs* @ *replicate* (*max k*1 *k*2) *False*)))
by (*auto simp*: *bdd-lookup-append*)
thus *?thesis* **by** (*rule exI*)
next
assume *?R*
then obtain *bs* **where** *O*: *bddh* (*length bs*) *r*1 ∧ *bddh* (*length bs*) *r*2 ∧ (∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup r*1 *bs*) (*bdd-lookup r*2 *bs*)) **unfolding** *IV*2 **by** *blast*
from *bddh-exists* **obtain** *k*1 *k*2 **where** *K*1: *bddh k*1 *l*1 **and** *K*2: *bddh k*2 *l*2 **by** *blast*
with *O* **have** *bddh* (*length bs* + *max k*1 *k*2) *l*1 **and** *bddh* (*length bs* + *max k*1 *k*2) *l*2 **and** *bddh* (*length bs* + *max k*1 *k*2) *r*1 **and** *bddh* (*length bs* + *max k*1 *k*2) *r*2
by (*simp add*: *bddh-ge*[*of length bs length bs* + *max k*1 *k*2] *bddh-ge*[*of k*1 *length bs* + *max k*1 *k*2] *bddh-ge*[*of k*2 *length bs* + *max k*1 *k*2])+
with *O* **have** *bddh* (*length* (*True* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*Branch l*1 *r*1) ∧ *bddh* (*length* (*True* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*Branch l*2 *r*2) ∧
(∃ *n*<*m*. *dist-nodes M n v* (*bdd-lookup* (*Branch l*1 *r*1) (*True* # *bs* @ *replicate* (*max k*1 *k*2) *False*)) (*bdd-lookup* (*Branch l*2 *r*2) (*True* # *bs* @ *replicate* (*max k*1 *k*2) *False*)))
by (*auto simp*: *bdd-lookup-append*)
thus *?thesis* **by** (*rule exI*)
qed

```
  next
    assume ∃ 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 ∨ ?R by (cases b) auto
  qed
  finally show ?case by simp
qed

lemma iter-wf: wf-tr M T ⟹ wf-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-tr M T ⟹ wf-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 ≤ length bss
  shows ∃ b bs. bss = b @ bs ∧ 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 ∃ b bs. as = b @ bs ∧ length b = n' by simp
    then obtain b bs where B: as = b @ bs ∧ length b = n' by blast
    with Suc Cons have a # as = (a # b) @ bs ∧ length (a # b) = n by simp
    thus ?thesis by blast
  qed simp
qed simp

lemma iter-dist-nodes:
  assumes wf-tr M T
  and wf-dfa M v
  and ∀ p q. dfa-is-node M q ∧ p < q ⟶ tr-lookup T q p = (∃ n < m. dist-nodes
M n v p q) and m > 0
  and dfa-is-node M q and p < q
  shows tr-lookup (snd (iter M T)) q p = (∃ n < Suc m. dist-nodes M n v p q)
proof −
  from assms obtain m' where M': m = Suc m' by (cases m) simp+
  have C: (¬ check-eq (fst M ! q) (fst M ! p) T) = (∃ n<m. dist-nodes M (Suc
n) v p q) proof
    assume ¬ check-eq (fst M ! q) (fst M ! p) T
    with assms have ∃ bs. bddh (length bs) (fst M ! q) ∧ bddh (length bs) (fst M
```

! p) $\wedge$ ($\exists\, n < m$. *dist-nodes* $M\ n\ v$ (*bdd-lookup* (*fst* M ! q) *bs*) (*bdd-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-all* (*is-alph v*) *bss* $\wedge$ *dfa-accepting* M (*dfa-steps* M (*bdd-lookup* (*fst* M ! q) *bs*) *bss*) $\neq$ *dfa-accepting* M (*dfa-steps* M (*bdd-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-accepting* M (*dfa-steps* M (*bdd-lookup* (*fst* M ! q) (*bs* @ *replicate v False*)) *bss*) $\neq$ *dfa-accepting* M (*dfa-steps* M (*bdd-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-accepting* M (*dfa-steps* M (*dfa-trans* $M\ q\ b'$) *bss*) $\neq$ *dfa-accepting* M (*dfa-steps* M (*dfa-trans* $M\ p\ b'$) *bss*) **by** (*auto simp*: *bdd-lookup-append dfa-trans-def*)
with X V **have** *is-alph v* $b' \wedge$ *dist-nodes* $M\ n\ v$ (*dfa-trans* $M\ p\ b'$) (*dfa-trans* $M\ q\ b'$) **by** (*auto simp*: *dist-nodes-def is-alph-def*)
hence *dist-nodes* M (*Suc n*) $v\ p\ q$ **by** (*auto simp*: *dist-nodes-suc*)
with X **show** $\exists\, n{<}m$. *dist-nodes* M (*Suc n*) $v\ p\ q$ **by** *auto*
next
assume $\exists\, n{<}m$. *dist-nodes* M (*Suc n*) $v\ p\ q$
hence $\exists\, bs.\ \exists\, n{<}m$. *is-alph v bs* $\wedge$ *dist-nodes* $M\ n\ v$ (*dfa-trans* $M\ p\ bs$) (*dfa-trans* $M\ q\ bs$) **by** (*auto simp*: *dist-nodes-suc*)
then obtain *bs* **where** X: $\exists\, n{<}m$. *is-alph v bs* $\wedge$ *dist-nodes* $M\ n\ v$ (*dfa-trans* $M\ p\ bs$) (*dfa-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-nodes* $M\ n\ v$ (*bdd-lookup* (*fst* M ! q) *bs*) (*bdd-lookup* (*fst* M ! p) *bs*)) **by** (*auto simp*: *dfa-trans-def dist-nodes-def*)
moreover from *assms* **have** *bdd-all* (*dfa-is-node* M) (*fst* M ! p) $\wedge$ *bdd-all* (*dfa-is-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-eq* (*fst* M ! q) (*fst* M ! p) T **by** (*auto simp*: *check-eq-dist-nodes*)
qed

from *assms* **have** *tr-lookup* (*snd* (*iter* M T)) $q\ p =$
(*if tr-lookup* $T\ q\ p$ *then True else* $\neg$ *check-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 ... = (*tr-lookup T q p* ∨ ¬ *check-eq* (*fst M* ! *q*) (*fst M* ! *p*) *T*) **by** *simp*
also from *assms C* **have** ... = ((∃ *n*<*m*. *dist-nodes M n v p q*) ∨ (∃ *n*<*m*. *dist-nodes M* (*Suc n*) *v p q*)) **by** *simp*
also have ... = (∃ *n* < *m*. *dist-nodes M n v p q* ∨ *dist-nodes M* (*Suc n*) *v p q*) **by** *auto*
also have ... = (∃ *n* < *Suc m*. *dist-nodes M n v p q*) **proof**
assume ∃ *n*<*m*. *dist-nodes M n v p q* ∨ *dist-nodes M* (*Suc n*) *v p q*
then obtain *n* **where** *D*: *dist-nodes M n v p q* ∨ *dist-nodes M* (*Suc n*) *v p q* **and** *N*: *n* < *m* **by** *blast*
moreover from *N* **have** *n* < *Suc m* **by** *simp*
ultimately show ∃ *n* < *Suc m*. *dist-nodes M n v p q* **by** (*elim disjE*) *blast*+
next
assume ∃ *n* < *Suc m*. *dist-nodes M n v p q*
then obtain *n* **where** *N*: *n* < *Suc m* **and** *D*: *dist-nodes M n v p q* **by** *blast*
from *N* **have** *n* < *m* ∨ *n* = *m* **by** *auto*
from *this D M'* **show** ∃ *n*<*m*. *dist-nodes M n v p q* ∨ *dist-nodes M* (*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 ∀ *p q*. *dfa-is-node M q* ∧ *p* < *q* ⟶ *tr-lookup T q p* = (∃ *n* < *m*. *dist-nodes M n v p q*) **and** *m* > 0
and *dfa-is-node M q* **and** *p* < *q*
shows *tr-lookup* (*fixpt M T*) *q p* = (∃ *n*. *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* = *snd* (*iter M T*)
show *?case* **proof** (*cases fst* (*iter M T*))
case *True*
{ fix *p' q'* **assume** *H*: *dfa-is-node M q'* ∧ *p'* < *q'*
with 1 **have** *tr-lookup ?T q' p'* = (∃ *n* < *Suc m*. *dist-nodes M n v p' q'*) **by** (*simp only*: *iter-dist-nodes*)
} hence 2: ∀ *p q*. *dfa-is-node M q* ∧ *p* < *q* ⟶ *tr-lookup ?T q p* = (∃ *n* < *Suc m*. *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)[*of Suc m*] *True*
ultimately have *tr-lookup* (*fixpt M ?T*) *q p* = (∃ *n*. *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*: *snd* (*iter M T*) = *T* **by** (*simp add*: *iter-def fold-map-idx-fst-snd-eq split-beta*)
have *C*: ⋀*m'*. ∀ *p q*. *dfa-is-node M q* ∧ *p* < *q* ⟶ *tr-lookup T q p* = (∃ *n* <

```
m' + m. dist-nodes M n v p q)
    proof −
      fix m' show ∀p q. dfa-is-node M q ∧ p < q ⟶ tr-lookup T q p = (∃n <
m' + m. 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 $\ldots = (\exists\, n < 1.\ \mathit{dist\text{-}nodes}\ M\ n\ v\ p'\ q')$ **by** *auto*
finally have $\mathit{tr\text{-}lookup}\ (\mathit{init\text{-}tr}\ M)\ q'\ p' = (\exists\, n{<}1.\ \mathit{dist\text{-}nodes}\ M\ n\ v\ p'\ q')$ **by** *simp*
} **hence** $\forall\, p\ q.\ \mathit{dfa\text{-}is\text{-}node}\ M\ q \wedge p < q \longrightarrow \mathit{tr\text{-}lookup}\ (\mathit{init\text{-}tr}\ M)\ q\ p = (\exists\, n{<}1.\ \mathit{dist\text{-}nodes}\ M\ n\ v\ p\ q)$ **by** *simp*
moreover note $H1\ H2$
ultimately have $\mathit{tr\text{-}lookup}\ (\mathit{fixpt}\ M\ (\mathit{init\text{-}tr}\ M))\ q\ p = (\exists\, n.\ \mathit{dist\text{-}nodes}\ M\ n\ v\ p\ q)$ **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* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool list list* ⇒ *nat option list*
where
mk-eqcl' [] *i j l T* = []
| *mk-eqcl'* (*x*#*xs*) *i j l T* = (*if tr-lookup T j i* ∨ *x* ≠ *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* ⇒ *nat list* ⇒ *nat* ⇒ *bool list list* ⇒ *nat list* × *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 ⋀*x k*. ⟦*x* ∈ *set xs*; *x* = *Some k*⟧ ⟹ *k* < *l*
and *x* ∈ *set* (*mk-eqcl' xs i j l T*) **and** *x* = *Some k*
shows *k* ≤ *l*
using *assms* **proof** (*induct xs arbitrary*: *j*)
case (*Cons y xs j*)
hence *x* = *y* ∨ *x* = *Some l* ∨ *x* ∈ *set* (*mk-eqcl' xs i* (*Suc j*) *l T*) **by** (*cases tr-lookup T j i* ∨ *y* ≠ *None*) *auto*
thus *?case* **proof** (*elim disjE*)
assume *x* = *y*
hence *x* ∈ *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 ⋀*x k*. ⟦*x* ∈ *set xs*; *x* = *Some k*⟧ ⟹ *k* < *l*

```
  and ⋀i'. ⟦i' < length xs; ¬ tr-lookup T (i' + j) i⟧ ⟹ xs ! i' = None
  and i < j and j' < length xs
  shows (mk-eqcl' xs i j l T ! j' = Some l) = (¬ tr-lookup T (j' + j) i)
using assms proof (induct xs arbitrary: j j')
  case (Cons x xs j)
  have I1:⋀i'. ⟦i' < length xs; ¬ tr-lookup T (i' + Suc j) i⟧ ⟹ xs ! i' = None
proof −
    fix i' assume H: i' < length xs ¬ tr-lookup T (i' + Suc j) i
    with Cons(3)[of Suc i'] show xs ! i' = None by simp
  qed
  have j' = 0 ∨ j' > 0 by auto
  thus ?case proof (elim disjE)
    assume j' > 0
    then obtain j'' where J: j' = Suc j'' by (cases j') simp+
    from Cons(1)[of Suc j j''] I1 Cons(2,4,5) J show ?thesis by simp
  next
    assume H: j' = 0
    with Cons(3)[of 0] have ¬ tr-lookup T j i ⟹ x = None by simp
    with Cons H show ?thesis by auto
  qed
qed simp

lemma mk-eqcl'-nth:
  assumes ⋀i' j' k. ⟦i' < length xs; j' < length xs; xs ! i' = Some k⟧ ⟹ (xs ! j'
= Some k) = (¬ tr-lookup T (i' + jj) (j' + jj))
  and ⋀a b c. ⟦a ≤ length T; b ≤ length T; c ≤ length T; ¬ tr-lookup T a b; ¬
tr-lookup T b c⟧ ⟹ ¬ tr-lookup T a c
  and length xs + jj = length T + 1
  and ⋀x k. ⟦x ∈ set xs; x = Some k⟧ ⟹ k < l
  and ⋀i'. ⟦i' < length xs; ¬ tr-lookup T (i' + jj) ii⟧ ⟹ xs ! i' = None
  and ii < jj
  and i < length xs and mk-eqcl' xs ii jj l T ! i = Some m
  and j < length xs
  shows (mk-eqcl' xs ii jj l T ! j = Some m) = (¬ tr-lookup T (i + jj) (j + jj))
using assms proof (induct xs arbitrary: jj i j)
  case Nil
  from Nil(7) have False by simp
  thus ?case by simp
next
  case (Cons y xs jj i j)
  show ?case proof (cases i)
    case 0
    show ?thesis proof (cases j)
      case 0
      with ‹i=0› Cons(9) show ?thesis by (simp add: tr-lookup-def)
    next
      case (Suc j')
      from 0 Cons(5,9) have 1: y = Some m ∧ m < l ∨ (y = None ∧ ¬ tr-lookup
T jj ii ∧ m = l) by (cases y, cases tr-lookup T jj ii, auto)
```

```
    thus ?thesis proof (elim disjE)
      assume H: y = Some m ∧ m < l
      from Suc have (mk-eqcl' (y # xs) ii jj l T ! j = Some m) = (mk-eqcl' xs
ii (Suc jj) l T ! j' = Some m) by simp
      also from H have ... = (xs ! j' = 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 ... = (¬ tr-lookup
T (i + jj) (j + jj)) by simp
      finally show ?thesis by simp
    next
      assume H: y = None ∧ ¬ tr-lookup T jj ii ∧ 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 ... = (¬ tr-lookup T (j' + Suc jj) ii) proof (rule mk-eqcl'-nth')
        from Cons(5) show ⋀x k. ⟦x ∈ set xs; x = Some k⟧ ⟹ k < l by simp
        show ⋀i'. ⟦i' < length xs; ¬ tr-lookup T (i' + Suc jj) ii⟧ ⟹ xs ! i' =
None proof −
          fix i' assume i' < length xs ¬ 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 ... = (¬ tr-lookup T (j + jj) ii ∧ ¬ tr-lookup T
(i + jj) ii) by (simp add: add.commute)
      also have ... = (¬ tr-lookup T (i + jj) (j + jj) ∧ ¬ tr-lookup T (i + jj)
ii) proof
        assume H': ¬ tr-lookup T (j + jj) ii ∧ ¬ tr-lookup T (i + jj) ii
        hence ¬ 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 ¬ tr-lookup T
(i + jj) (j + jj) ∧ ¬ tr-lookup T (i + jj) ii by simp
      next
        assume H': ¬ tr-lookup T (i + jj) (j + jj) ∧ ¬ tr-lookup T (i + jj) ii
        hence ¬ 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 ¬ tr-lookup T
(j + jj) ii ∧ ¬ tr-lookup T (i + jj) ii by simp
      qed
      also from 0 H have ... = (¬ 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 ≤ l proof (rule mk-eqcl'-bound)
```

from *Cons*(5) **show** $\bigwedge x\ k.\ [\![x \in set\ (y\ \#\ xs);\ x = Some\ k]\!] \Longrightarrow k < l$ **by** *simp*

from *Cons*(8) **have** $i < length\ (mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T)$ **by** (*simp add*: *mk-eqcl'-len*)

with *Cons*(9) **have** $\exists\, i < length\ (mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T).\ mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T\ !\ i = Some\ m$ **by** *blast*

thus $Some\ m \in set\ (mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T)$ **by** (*simp only*: *in-set-conv-nth*)

show $Some\ m = Some\ m$ **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*: $(y\ \#\ xs)\ !\ i = Some\ m$ **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* $\vee\ a \neq None$, *simp+*)

qed *simp*

from 0 *H* **have** $(mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T\ !\ j = Some\ m) = ((y\#xs)\ !\ j = Some\ m)$ **by** (*cases tr-lookup T jj ii* $\vee\ y \neq None$) *simp+*

also from *Cons*(8,10) *I* **have** $\ldots = (\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj))$ **by** (*rule Cons*(2))

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

next

assume *H*: $m = l$

from *Cons*(5,6,7,8) **have** $(mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T\ !\ i = Some\ l) = (\neg\ tr\text{-}lookup\ T\ (i + jj)\ ii)$ **by** (*rule mk-eqcl'-nth'*)

with *H Cons*(9) **have** *I*: $\neg\ tr\text{-}lookup\ T\ (i + jj)\ ii$ **by** *simp*

with 0 *H Cons*(5) **have** $(mk\text{-}eqcl'\ (y\ \#\ xs)\ ii\ jj\ l\ T\ !\ j = Some\ m) = (\neg\ tr\text{-}lookup\ T\ (j + jj)\ ii \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii \wedge y = None)$ **by** *auto*

also from *Cons*(6)[*of* 0] 0 **have** $\ldots = (\neg\ tr\text{-}lookup\ T\ (j + jj)\ ii \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii)$ **by** *auto*

also have $\ldots = (\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj) \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii)$ **proof**

assume *H'*: $\neg\ tr\text{-}lookup\ T\ (j + jj)\ ii \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii$

hence $\neg\ tr\text{-}lookup\ T\ ii\ (j + jj)$ **by** (*auto simp*: *tr-lookup-def*)

with *H' Cons*(3)[*of i + jj ii j + jj*] *Cons*(4,7,8,10) **show** $\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj) \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii$ **by** *simp*

next

assume *H'*: $\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj) \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii$

hence $\neg\ tr\text{-}lookup\ T\ (j + jj)\ (i + jj)$ **by** (*auto simp*: *tr-lookup-def*)

with *H' Cons*(3)[*of j + jj i + jj ii*] *Cons*(4,7,8,10) **show** $\neg\ tr\text{-}lookup\ T\ (j + jj)\ ii \wedge \neg\ tr\text{-}lookup\ T\ (i + jj)\ ii$ **by** *simp*

qed

also from *I* **have** $\ldots = (\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj))$ **by** *simp*

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

qed

next

case (*Suc j'*)

hence (*mk-eqcl′* (*y* # *xs*) *ii jj l T* ! *j* = *Some m*) = (*mk-eqcl′ xs ii* (*Suc jj*) *l T* ! *j′* = *Some m*) **by** *simp*
also have ... = (¬ *tr-lookup T* (*i′* + *Suc jj*) (*j′* + *Suc jj*)) **proof** (*rule Cons*(1))
show ⋀*i′ j′ k*. ⟦*i′* < *length xs*; *j′* < *length xs*; *xs* ! *i′* = *Some k*⟧ ⟹ (*xs* ! *j′* = *Some k*) = (¬ *tr-lookup T* (*i′* + *Suc jj*) (*j′* + *Suc jj*)) **proof** −
fix *i′ j′ k* **assume** *i′* < *length xs j′* < *length xs xs* ! *i′* = *Some k*
with *Cons*(2)[*of Suc i′ Suc j′ k*] **show** (*xs* ! *j′* = *Some k*) = (¬ *tr-lookup T* (*i′* + *Suc jj*) (*j′* + *Suc jj*)) **by** *simp*
qed
from *Cons*(3) **show** ⋀*a b c*. ⟦*a* ≤ *length T*; *b* ≤ *length T*; *c* ≤ *length T*; ¬ *tr-lookup T a b*; ¬ *tr-lookup T b c* ⟧ ⟹ ¬ *tr-lookup T a c* **by** *blast*
from *Cons*(4) **show** *length xs* + *Suc jj* = *length T* + 1 **by** *simp*
from *Cons*(5) **show** ⋀*x k*. ⟦*x* ∈ *set xs*; *x* = *Some k*⟧ ⟹ *k* < *l* **by** *simp*
show ⋀*i′*. ⟦*i′* < *length xs*; ¬ *tr-lookup T* (*i′* + *Suc jj*) *ii*⟧ ⟹ *xs* ! *i′* = *None* **proof** −
fix *i′* **assume** *i′* < *length xs* ¬ *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*(8) ‹*i*=*Suc i′*› **show** *i′* < *length xs* **by** *simp*
from *Cons*(9) ‹*i*=*Suc i′*› **show** *mk-eqcl′ xs ii* (*Suc jj*) *l T* ! *i′* = *Some m* **by** *simp*
from *Cons*(10) *Suc* **show** *j′* < *length xs* **by** *simp*
qed
also from *Suc* ‹*i*=*Suc i′*› **have** ... = (¬ *tr-lookup T* (*i* + *jj*) (*j* + *jj*)) **by** *simp*
finally show *?thesis* **by** *simp*
qed
qed
qed

lemma *mk-eqcl′-Some*:
assumes *i* < *length xs* **and** *xs* ! *i* ≠ *None*
shows *mk-eqcl′ xs ii j l T* ! *i* = *xs* ! *i*
using *assms* **proof** (*induct xs arbitrary*: *j i*)
case (*Cons y xs j i*)
thus *?case* **by** (*cases i*) *auto*
qed *simp*

lemma *mk-eqcl′-Some*2:
assumes *i* < *length xs*
and *k* < *l*
shows (*mk-eqcl′ xs ii j l T* ! *i* = *Some k*) = (*xs* ! *i* = *Some k*)
using *assms* **proof** (*induct xs arbitrary*: *j i*)
case (*Cons y xs j i*)
thus *?case* **by** (*cases i*) *auto*
qed *simp*

lemma *mk-eqcl-fst-Some*:
assumes $i < length\ xs$ **and** $k < length\ zs$
shows $(fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i = k) = (xs\ !\ i = Some\ k)$
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′-Some*2)+
next
case (3 *l xs zs ii T i*)
thus *?case* **by** (*cases i*) (*simp add*: *split-beta*)+
qed *simp*

lemma *mk-eqcl-len-snd*:
$length\ zs \leq length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T))$
by (*induct xs zs i T rule*: *mk-eqcl.induct*) (*simp add*: *split-beta*)+

lemma *mk-eqcl-len-fst*:
$length\ (fst\ (mk\text{-}eqcl\ xs\ zs\ i\ T)) = length\ xs$
by (*induct xs zs i T rule*: *mk-eqcl.induct*) (*simp add*: *split-beta mk-eqcl′-len*)+

lemma *mk-eqcl-set-snd*:
assumes $i \notin set\ zs$
and $j > i$
shows $i \notin set\ (snd\ (mk\text{-}eqcl\ xs\ zs\ j\ T))$
using *assms* **by** (*induct xs zs j T rule*: *mk-eqcl.induct*) (*auto simp*: *split-beta*)

lemma *mk-eqcl-snd-mon*:
assumes $\bigwedge j1\ j2.\ [\![j1 < j2; j2 < length\ zs]\!] \Longrightarrow zs\ !\ j1 < zs\ !\ j2$
and $\bigwedge x.\ x \in set\ zs \Longrightarrow x < i$
and $j1 < j2$ **and** $j2 < length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T))$
shows $snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T)\ !\ j1 < snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T)\ !\ j2$
using *assms* **proof** (*induct xs zs i T rule*: *mk-eqcl.induct*)
case (2 *xs zs i T*)
have $\bigwedge j1\ j2.\ [\![j1 < j2; j2 < length\ (zs\ @\ [i])]\!] \Longrightarrow (zs\ @\ [i])\ !\ j1 < (zs\ @\ [i])\ !\ j2$ **proof** −
fix *j1 j2* **assume** *H*: $j1 < j2\ j2 < length\ (zs\ @\ [i])$
hence $j2 < length\ zs \vee j2 = length\ zs$ **by** *auto*
from *this H* 2 **show** $(zs\ @\ [i])\ !\ j1 < (zs\ @\ [i])\ !\ j2$ **by** (*elim disjE*) (*simp add*: *nth-append*)+
qed moreover
have $\bigwedge x.\ x \in set\ (zs\ @\ [i]) \Longrightarrow x < Suc\ i$ **proof** −
fix *x* **assume** $x \in set\ (zs\ @\ [i])$
hence $x \in set\ zs \vee x = i$ **by** *auto*
with 2(3)[*of x*] **show** $x < Suc\ i$ **by** *auto*
qed moreover
note 2(4) **moreover**
from 2(5) **have** $j2 < length\ (snd\ (mk\text{-}eqcl\ (mk\text{-}eqcl′\ xs\ i\ (Suc\ i)\ (length\ zs)\ T)\ (zs\ @\ [i])\ (Suc\ i)\ T))$ **by** (*simp add*: *split-beta*)
ultimately have $snd\ (mk\text{-}eqcl\ (mk\text{-}eqcl′\ xs\ i\ (Suc\ i)\ (length\ zs)\ T)\ (zs\ @\ [i])\ (Suc\ i)\ T)\ !\ j1 < snd\ (mk\text{-}eqcl\ (mk\text{-}eqcl′\ xs\ i\ (Suc\ i)\ (length\ zs)\ T)\ (zs\ @\ [i])$

(*Suc i*) *T*) ! *j*2 **by** (*rule* 2(1))
thus *?case* **by** (*simp add*: *split-beta*)
next
case (3 *l xs zs i T*)
note 3(2) **moreover**
have $\bigwedge x.\ x \in set\ zs \Longrightarrow x < Suc\ i$ **proof** −
fix *x* **assume** $x \in set\ zs$
with 3(3)[*of x*] **show** $x < Suc\ i$ **by** *simp*
qed moreover
note 3(4) **moreover**
from 3(5) **have** $j2 < length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ (Suc\ i)\ T))$ **by** (*simp add*: *split-beta*)
ultimately have *snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j*1 < *snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j*2 **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 $\bigwedge x\ k.\ [\![x \in set\ xs;\ x = Some\ k]\!] \Longrightarrow k < length\ zs$
and *x* ∈ *set* (*fst* (*mk-eqcl xs zs ii T*))
shows *x* < *length* (*snd* (*mk-eqcl xs zs ii T*))
using *assms* **proof** (*induct xs zs ii T rule*: *mk-eqcl.induct*)
case (2 *xs zs i T*)
hence *x* = *length zs* ∨ *x* ∈ *set* (*fst* (*mk-eqcl* (*mk-eqcl′ xs i* (*Suc i*) (*length zs*) *T*) (*zs* @ [*i*]) (*Suc i*) *T*)) **by** (*auto simp*: *split-beta*)
thus *?case* **proof** (*elim disjE*)
assume *x* = *length zs*
hence *x* < *length* (*zs* @ [*i*]) **by** *simp*
also have ... ≤ *length* (*snd* (*mk-eqcl* (*mk-eqcl′ xs i* (*Suc i*) (*length zs*) *T*) (*zs* @ [*i*]) (*Suc i*) *T*)) **by** (*simp only*: *mk-eqcl-len-snd*)
finally show *?thesis* **by** (*simp add*: *split-beta*)
next
assume *H*: *x* ∈ *set* (*fst* (*mk-eqcl* (*mk-eqcl′ xs i* (*Suc i*) (*length zs*) *T*) (*zs* @ [*i*]) (*Suc i*) *T*))
have $\bigwedge x\ k.\ [\![x \in set\ (mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T);\ x = Some\ k]\!] \Longrightarrow k < length\ (zs\ @\ [i])$ **proof** −
fix *x k* **assume** *H′*: *x* ∈ *set* (*mk-eqcl′ xs i* (*Suc i*) (*length zs*) *T*) *x* = *Some k*
{ **fix** *x′ k′* **assume** *x′* ∈ *set xs x′* = *Some k′*
with 2 **have** *k′* < *length zs* **by** *simp*
} **from** *this H′* **have** *k* ≤ *length zs* **by** (*rule mk-eqcl′-bound*)
thus *k* < *length* (*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 $x = l \vee x \in set\ (fst\ (mk\text{-}eqcl\ xs\ zs\ (Suc\ i)\ T))$ **by** (*auto simp*: *split-beta*)
thus *?case* **proof** (*elim disjE*)
assume $x = l$
with 3 **have** $x < length\ zs$ **by** *simp*
also from 3 **have** $\ldots \leq length\ (snd\ (mk\text{-}eqcl\ (Some\ l\ \#\ xs)\ zs\ i\ T))$ **by** (*simp only*: *mk-eqcl-len-snd*)
finally show *?thesis* **by** *simp*
next
assume $x \in set\ (fst\ (mk\text{-}eqcl\ xs\ zs\ (Suc\ i)\ T))$
with 3 **have** $x < length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ (Suc\ i)\ T))$ **by** *simp*
thus *?thesis* **by** (*simp add*: *split-beta*)
qed
qed *simp*

lemma *mk-eqcl-fst-snd*:
assumes $\bigwedge i.\ i < length\ zs \Longrightarrow zs\ !\ i < length\ xs + ii \wedge (zs\ !\ i \geq ii \longrightarrow xs\ !\ (zs\ !\ i - ii) = Some\ i)$
and $\bigwedge j1\ j2.\ [\![j1 < j2; j2 < length\ zs]\!] \Longrightarrow zs\ !\ j1 < zs\ !\ j2$
and $\bigwedge z.\ z \in set\ zs \Longrightarrow z < ii$
and $i < length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T))$
and $length\ xs + ii \leq length\ T + 1$
shows $snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i < length\ (fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)) + ii \wedge (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i \geq ii \longrightarrow\ fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i - ii) = i)$
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 $\bigwedge i'.\ i' < length\ (zs\ @\ [i]) \Longrightarrow (zs\ @\ [i])\ !\ i' < length\ (mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T) + Suc\ i \wedge$
$(Suc\ i \leq (zs\ @\ [i])\ !\ i' \longrightarrow mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T\ !\ ((zs\ @\ [i])\ !\ i' - Suc\ i) = Some\ i')$ **proof** −
fix i' **assume** $i' < length\ (zs\ @\ [i])$
hence $i' < length\ zs \vee i' = length\ zs$ **by** *auto*
thus $(zs\ @\ [i])\ !\ i' < length\ (mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T) + Suc\ i \wedge (Suc\ i \leq (zs\ @\ [i])\ !\ i' \longrightarrow mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T\ !\ ((zs\ @\ [i])\ !\ i' - Suc\ i) = Some\ i')$
proof (*elim disjE*)
assume *H*: $i' < length\ zs$
with 2(2) **have** *I*: $zs\ !\ i' < length\ (None\ \#\ xs) + i \wedge (i \leq zs\ !\ i' \longrightarrow (None\ \#\ xs)\ !\ (zs\ !\ i' - i) = Some\ i')$ **by** *simp*
with *H* **have** *G*1: $(zs\ @\ [i])\ !\ i' < length\ (mk\text{-}eqcl'\ xs\ i\ (Suc\ i)\ (length\ zs)\ T) + Suc\ i$ **by** (*auto simp*: *mk-eqcl'-len nth-append*)
{ assume *H'*: $Suc\ i \leq (zs\ @\ [i])\ !\ i'$
then obtain *k* **where** *K*: $(zs\ @\ [i])\ !\ i' - i = Suc\ k$ **by** (*cases* $(zs\ @\ [i])\ !\ i' - i$) *simp+*

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

```
      also have ... ≤ length (snd (mk-eqcl (mk-eqcl' xs i (Suc i) (length zs) T)
(zs @ [i]) (Suc i) T)) by (simp only: mk-eqcl-len-snd)
      also have ... = length (snd (mk-eqcl (None # xs) zs i T)) by (simp add:
split-beta)
      finally have K': k < length (snd (mk-eqcl (None # xs) zs i T)) by simp
       from 2(3,4) H' this have snd (mk-eqcl (None # xs) zs i T) ! j < 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 ⋀j. j < length zs ⟹ zs ! j < length xs + Suc i ∧ (Suc i ≤ zs ! j ⟶ xs
! (zs ! j − Suc i) = Some j) proof −
   fix j assume H: j < length zs
   with 3(2)[of j] have I: zs ! j < length (Some l # xs) + i ∧ (i ≤ zs ! j ⟶
(Some l # xs) ! (zs ! j − i) = Some j) by simp
   hence G1: zs ! j < length xs + Suc i and G2: i ≤ zs ! j ⟶ (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 *G*1 **show** *zs* ! *j* < *length xs* + *Suc i* ∧ (*Suc i* ≤ *zs* ! *j* ⟶ *xs* ! (*zs* ! *j* − *Suc i*) = *Some j*) **by** *simp*
qed
moreover note 3(3)
moreover have ⋀*z*. *z* ∈ *set zs* ⟹ *z* < *Suc i* **proof** −
fix *z* **assume** *z* ∈ *set zs*
with 3(4)[*of z*] **show** *z* < *Suc i* **by** *simp*
qed
moreover from 3(5) 1 **have** *j* < *length* (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*)) **by** *simp*
moreover from 3 **have** *length xs* + *Suc i* ≤ *length T* + 1 **by** *simp*
ultimately have *IV*: *snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j* < *length* (*fst* (*mk-eqcl xs zs* (*Suc i*) *T*)) + *Suc i* ∧
(*Suc i* ≤ *snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j* ⟶ *fst* (*mk-eqcl xs zs* (*Suc i*) *T*) ! (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j* − *Suc i*) = *j*) **by** (*rule* 3(1))
with 1 **have** *G*1: *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* < *length* (*fst* (*mk-eqcl* (*Some l* # *xs*) *zs i T*)) + *i* **by** (*simp add*: *split-beta mk-eqcl-len-fst*)
{ assume *i* ≤ *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j*
hence *i* = *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* ∨ *i* < *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* **by** *auto*
hence *fst* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! (*snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* − *i*) = *j* **proof** (*elim disjE*)
assume *H*: *i* = *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j*
with 3 1 **have** ∃*j* < *length* (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*)). *snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j* = *i* **by** *auto*
hence *T*1: *i* ∈ *set* (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*)) **by** (*simp only*: *in-set-conv-nth*)
from 3(4) **have** *i* ∉ *set zs* **by** *auto*
hence *i* ∉ *set* (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*)) **by** (*simp add*: *mk-eqcl-set-snd*)
with *T*1 **show** *?thesis* **by** *simp*
next
assume *H*: *i* < *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j*
from *H* **obtain** *k* **where** *K*: *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* − *i* = *Suc k* **by** (*cases snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* − *i*) *simp*+
hence *K*′: *snd* (*mk-eqcl* (*Some l* # *xs*) *zs i T*) ! *j* − *Suc i* = *k* **by** *simp*
from 1 *H IV* **have** *fst* (*mk-eqcl xs zs* (*Suc i*) *T*) ! (*snd* (*mk-eqcl xs zs* (*Suc i*) *T*) ! *j* − *Suc i*) = *j* **by** *simp*
with *K K*′ **show** *?thesis* **by** (*simp add*: *split-beta*)
qed
} with *G*1 **show** *?case* **by** *simp*
qed

lemma *mk-eqcl-fst-nth*:
assumes ⋀*i j k*. ⟦*i* < *length xs*; *j* < *length xs*; *xs* ! *i* = *Some k*⟧ ⟹ (*xs* ! *j* = *Some k*) = (¬ *tr-lookup T* (*i* + *ii*) (*j* + *ii*))
and ⋀*a b c*. ⟦*a* ≤ *length T*; *b* ≤ *length T*; *c* ≤ *length T*; ¬ *tr-lookup T a b*; ¬ *tr-lookup T b c*⟧ ⟹ ¬ *tr-lookup T a c*
and ⋀*x k*. ⟦*x* ∈ *set xs*; *x* = *Some k*⟧ ⟹ *k* < *length zs*
and *length xs* + *ii* = *length T* + 1
and *i* < *length xs* **and** *j* < *length xs*

shows (*fst* (*mk-eqcl xs zs ii T*) ! *i* = *fst* (*mk-eqcl xs zs ii T*) ! *j*) = (¬ *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 ii T*) **thus** *?case* **by** *simp*
next
case (2 *xs zs ii T*)
{ **fix** *i j* **assume** *H*: *i* < *j j* < *length* (*None* # *xs*)
then obtain *j'* **where** *J*: *j* = *Suc j'* **by** (*cases j*) *simp+*
have (*fst* (*mk-eqcl* (*None* # *xs*) *zs ii T*) ! *i* = *fst* (*mk-eqcl* (*None* # *xs*) *zs ii T*) ! *j*) = (¬ *tr-lookup T* (*i* + *ii*) (*j* + *ii*)) **proof** (*cases i*)
case 0
with *J* **have** (*fst* (*mk-eqcl* (*None* # *xs*) *zs ii T*) ! *i* = *fst* (*mk-eqcl* (*None* # *xs*) *zs ii T*) ! *j*) = (*fst* (*mk-eqcl* (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T*) (*zs* @ [*ii*]) (*Suc ii*) *T*) ! *j'* = *length zs*)
by (*auto simp add*: *split-beta*)
also from *H J* **have** ... = (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T* ! *j'* = *Some* (*length zs*)) **by** (*simp add*: *mk-eqcl-fst-Some mk-eqcl'-len*)
also have ... = (¬ *tr-lookup T* (*j'* + *Suc ii*) *ii*) **proof** −
have ⋀*x k*. ⟦*x* ∈ *set xs*; *x* = *Some k*⟧ ⟹ *k* < *length zs* **proof** −
fix *x k* **assume** *x* ∈ *set xs x* = *Some k*
with 2(4)[*of x k*] **show** *k* < *length zs* **by** *simp*
qed moreover
have ⋀*i'*. ⟦*i'* < *length xs*; ¬ *tr-lookup T* (*i'* + *Suc ii*) *ii*⟧ ⟹ *xs* ! *i'* = *None* **proof** −
fix *i'* **assume** *H*: *i'* < *length xs* ¬ *tr-lookup T* (*i'* + *Suc ii*) *ii*
{ **assume** *H'*: *xs* ! *i'* ≠ *None*
then obtain *k* **where** *xs* ! *i'* = *Some k* **by** (*cases xs* ! *i'*) *simp+*
with 2(2)[*of Suc i' 0 k*] *H* **have** *False* **by** *simp*
} **thus** *xs* ! *i'* = *None* **by** (*cases xs* ! *i'*) *simp+*
qed moreover
from *H J* **have** *ii* < *Suc ii j'* < *length xs* **by** *simp+*
ultimately show *?thesis* **by** (*rule mk-eqcl'-nth'*)
qed
also from *J* 0 **have** ... = (¬ *tr-lookup T* (*i* + *ii*) (*j* + *ii*)) **by** (*auto simp*: *tr-lookup-def*)
finally show *?thesis* **by** *simp*
next
case (*Suc i'*)
have ⋀*i j k*. ⟦*i* < *length* (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T*); *j* < *length* (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T*); *mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T* ! *i* = *Some k*⟧
⟹ (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T* ! *j* = *Some k*) = (¬ *tr-lookup T* (*i* + *Suc ii*) (*j* + *Suc ii*)) **proof** −
fix *i j k* **assume** *H*: *i* < *length* (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T*) *j* < *length* (*mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T*) *mk-eqcl' xs ii* (*Suc ii*) (*length zs*) *T* ! *i* = *Some k*
{ **fix** *i' j' k* **assume** *i'* < *length xs j'* < *length xs xs* ! *i'* = *Some k*
with 2(2)[*of Suc i' Suc j' k*] **have** (*xs* ! *j'* = *Some k*) = (¬ *tr-lookup T* (*i'* + *Suc ii*) (*j'* + *Suc ii*)) **by** *simp*

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

definition *min-dfa* :: *dfa* ⇒ *dfa* **where**
min-dfa = (λ(*bd*, *as*). *let* (*os*, *ns*) = *mk-eqcl* (*replicate* (*length bd*) *None*) [] 0 (*fixpt* (*bd*, *as*) (*init-tr* (*bd*, *as*))) *in*
(*map* (λ*p*. *bdd-map* (λ*q*. *os* ! *q*) (*bd* ! *p*)) *ns*, *map* (λ*p*. *as* ! *p*) *ns*))

definition *eq-nodes* :: *dfa* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
eq-nodes = (λ*M v p q*. ¬ (∃ *n*. *dist-nodes M n v p q*))

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

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

have (*fst ?M* ! *p* = *fst ?M* ! *q*) = (¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *p q*) **proof** −
{ fix *i j k* **assume** *H*: *i* < *length* (*replicate* (*length* (*fst M*)) *None*) *j* < *length* (*replicate* (*length* (*fst M*)) *None*) *replicate* (*length* (*fst M*)) *None* ! *i* = *Some k*
hence (*replicate* (*length* (*fst M*)) *None* ! *j* = *Some k*) = (¬ *tr-lookup* (*fixpt M* (*init-tr M*)) (*i* + 0) (*j* + 0)) **by** *simp*
}
moreover
have ⋀*a b c*. ⟦*a* ≤ *length* (*fixpt M* (*init-tr M*)); *b* ≤ *length* (*fixpt M* (*init-tr M*)); *c* ≤ *length* (*fixpt M* (*init-tr M*)); ¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *a b*; ¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *b c*⟧
⟹ ¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *a c* **proof** −
fix *a b c* **assume** *H'*: *a* ≤ *length* (*fixpt M* (*init-tr M*)) *b* ≤ *length* (*fixpt M* (*init-tr M*)) *c* ≤ *length* (*fixpt M* (*init-tr M*)) ¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *a b*
¬ *tr-lookup* (*fixpt M* (*init-tr M*)) *b c*
{ fix *q* **assume** *H''*: *q* ≤ *length* (*fixpt M* (*init-tr M*))
from *assms* **have** *length* (*fst M*) > 0 **by** (*simp add*: *wf-dfa-def*)
then obtain *m* **where** *M*: *length* (*fst M*) = *Suc m* **by** (*cases length* (*fst M*)) *simp*+

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

lemma *mk-eqcl-fixpt-fst-snd-nth*:
assumes $i < length\ (snd\ (mk\text{-}eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\text{-}tr\ M))))$
and *wf-dfa M v*
shows $snd\ (mk\text{-}eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\text{-}tr\ M)))\ !\ i < length\ (fst\ (mk\text{-}eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\text{-}tr\ M)))) \wedge$
$fst\ (mk\text{-}eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\text{-}tr\ M)))\ !\ (snd\ (mk\text{-}eqcl\ (replicate\ (length\ (fst\ M))\ None)\ []\ 0\ (fixpt\ M\ (init\text{-}tr\ M)))\ !\ i) = i$
(**is** $snd\ ?M\ !\ i < length\ (fst\ ?M) \wedge fst\ ?M\ !\ (snd\ ?M\ !\ i) = i$)
proof −
have $\bigwedge i.\ i < length\ [] \Longrightarrow []\ !\ i < length\ (replicate\ (length\ (fst\ M))\ None) + 0 \wedge (0 \leq []\ !\ i \longrightarrow replicate\ (length\ (fst\ M))\ None\ !\ ([]\ !\ i - 0) = Some\ i)$ **by** *simp*
moreover have $\bigwedge j1\ j2.\ [\![j1 < j2;\ j2 < length\ []]\!] \Longrightarrow []\ !\ j1 < []\ !\ j2$ **by** *simp*
moreover have $\bigwedge z.\ z \in set\ [] \Longrightarrow z < 0$ **by** *simp*
moreover note *assms*(1)
moreover have $length\ (replicate\ (length\ (fst\ M))\ None) + 0 \leq length\ (fixpt\ M\ (init\text{-}tr\ M)) + 1$ **proof** −

have *WF*: *wf-tr M* (*fixpt M* (*init-tr M*)) **by** (*simp only*: *init-tr-wf fixpt-wf*)
from *assms* **have** *length* (*fst M*) > 0 **by** (*simp add*: *wf-dfa-def*)
then obtain *m* **where** *M*:*length* (*fst M*) = *Suc m* **by** (*cases length* (*fst M*)) *simp*+
hence *M'*: *m* = *length* (*fst M*) − 1 **by** *simp*
with *WF* **have** *length* (*fixpt M* (*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 ∧ (0 ≤ *snd ?M* ! *i* ⟶ *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*: ¬ *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*) ≠ *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*)) ≠ *dfa-accepting M* (*dfa-steps M q* (*bs* # *w*)) **by** *simp*+
hence ¬ *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** *I*1: *?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** *I*2: *bddh* (*length bs*) (*fst M* ! *?q*) **by** (*simp add*: *mk-eqcl-len-fst wf-dfa-def list-all-iff is-alph-def*)
from *I*1 **have** *I*3: *dfa-is-node M ?q* **by** (*simp add*: *mk-eqcl-len-fst dfa-is-node-def*)
with *assms I*1 **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 I*3 **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*) [] 0 (*fixpt M* (*init-tr M*))) ! *p*) *w* =
fst (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 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*) [] 0 (*fixpt M* (*init-tr M*)))) > 0 ∧
fst (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*))) ! 0 = 0 ∧ *snd* (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*))) ! 0 = 0
(**is** *length* (*snd ?M*) > 0 ∧ *fst ?M* ! 0 = 0 ∧ *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 ... ≥ *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 ⟹ *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*) [] 0 (*fixpt M* (*init-tr M*))

{ fix *x* **assume** *x* ∈ *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* (λ*q*. *fst ?M* ! *q*) (*fst M* ! (*snd ?M* ! *i*)) **by** (*simp add*: *split-beta min-dfa-def*)
with *BH* **have** *G*1: *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* (λ*q*. *fst ?M* ! *q*) (*fst M* ! (*snd ?M* ! *i*))) **by** (*simp add*: *bdd-all-bdd-map*)
with *G*1 *BI I* **have** *bddh v x* ∧ *bdd-all* (*dfa-is-node* (*min-dfa M*)) *x* **by** *simp*
}
hence *G*: *list-all* (*bddh v*) *bd* ∧ *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 ∧ *fst ?M* ! 0 = 0 ∧ *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* = *snd ?M* ! (*fst ?M* ! *dfa-steps M* 0 *w*)

from *assms* **have** *N*: *dfa-is-node M* (*dfa-steps M* 0 *w*) **by** (*simp add*: *dfa-steps-is-node dfa-startnode-is-node*)
with *assms* **have** *I*: *?q* < *length* (*fst ?M*) *fst ?M* ! *?q* = *fst ?M* ! *dfa-steps M* 0 *w* **by** (*simp add*: *mk-eqcl-fixpt-fst-bound mk-eqcl-fixpt-fst-snd-nth*)+
hence *dfa-is-node M ?q* **by** (*simp add*: *mk-eqcl-len-fst dfa-is-node-def*)
with *assms N I* **have** *EQ*: *eq-nodes M v* (*dfa-steps M* 0 *w*) *?q* **by** (*simp add*: *mk-eqcl-fixpt-fst-nth*[*symmetric*])
have *A*: *dfa-accepting M* (*dfa-steps M* 0 *w*) = *dfa-accepting M ?q* **proof** (*rule ccontr*)
assume *H*: *dfa-accepting M* (*dfa-steps M* 0 *w*) ≠ *dfa-accepting M ?q*
hence *dist-nodes M* 0 *v* (*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** *dfa-accepts* (*min-dfa M*) *w* = *snd* (*min-dfa M*) ! (*fst ?M* ! *dfa-steps M* 0 *w*) **by** (*simp add*: *accepts-def dfa-accepting-def*)
also from *WF DN* **have** ... = *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** ... = *dfa-accepts M w* **by** (*simp add*: *accepts-def*)

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

4 NFAs

type-synonym *nbddtable* = *bool list bdd list*
type-synonym *nfa* = *nbddtable* × *astate*

definition
nfa-is-node :: *nfa* ⇒ *bool list* ⇒ *bool* **where**
nfa-is-node A = (λ*qs*. *length qs* = *length* (*fst A*))

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

definition

```
  set-of-bv :: bool list ⇒ nat set where
  set-of-bv bs = {i. i < length bs ∧ bs ! i}

fun
  bv-or :: bool list ⇒ bool list ⇒ bool list
where
  bv-or [] [] = [] |
  bv-or (x # xs) (y # ys) = (x ∨ y) # (bv-or xs ys)

lemma bv-or-nth:
  assumes length l = length r
  assumes i < length l
  shows bv-or l r ! i = (l ! i ∨ r ! i)
using assms proof (induct l r arbitrary: i rule: bv-or.induct)
  case (2 xx xss yy yss ii)
  have ii = 0 ∨ 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-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-of-bv (bv-or p q) = set-of-bv p ∪ set-of-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: ⟦nfa-is-node A p; nfa-is-node A q⟧ ⟹ nfa-is-node A (bv-or
p q)
  by (simp add: bv-or-length nfa-is-node-def)

fun subsetbdd
where
  subsetbdd [] [] bdd = bdd
| subsetbdd (bdd' # bdds) (b # bs) bdd =
    (if b then subsetbdd bdds bs (bdd-binop bv-or bdd bdd') else subsetbdd bdds bs
bdd)

definition
  nfa-emptybdd :: nat ⇒ bool list bdd where
  nfa-emptybdd n = Leaf (replicate n False)

lemma bdd-all-is-node-subsetbdd:
```

assumes *list-all* (*bdd-all* (*nfa-is-node A*)) (*fst A*)
and *nfa-is-node A q*
shows *bdd-all* (*nfa-is-node A*) (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*)))
proof −
{ fix *bdds* :: *bool list bdd list* **and** *q* :: *bool list* **and** *bd* :: *bool list bdd* **and** *n*
assume *list-all* (*bdd-all* (λ*l*. *length l* = *n*)) *bdds* **and** *bdd-all* (λ*l*. *length l* = *n*) *bd* **and** *length bdds* = *length q*
hence *bdd-all* (λ*l*. *length l* = *n*) (*subsetbdd bdds q bd*) **by** (*induct bdds q bd rule*: *subsetbdd.induct*) (*simp add*: *bdd-all-bdd-binop*[*of* λ*l*. *length l* =*n* - λ*l*. *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 *length bdds* = *length q*
and ∀ *x* ∈ *set bdds*. *bddh* (*length ws*) *x*
and *bddh* (*length ws*) *obdd*
and ⋀*bs w*. ⟦*bs* ∈ *set bdds*; *length w* = *length ws*⟧ ⟹ *length* (*bdd-lookup bs w*) = *c*
and ⋀*w*. *length w* = *length ws* ⟹ *length* (*bdd-lookup obdd w*) = *c*
and *a* < *c*
shows *bdd-lookup* (*subsetbdd bdds q obdd*) *ws* ! *a* = ((∃ *i* < *length q*. *q* ! *i* ∧ *bdd-lookup* (*bdds* ! *i*) *ws* ! *a*) ∨ *bdd-lookup 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* =
((∃ *i*<*length xs*. *xs* ! *i* ∧ *bdd-lookup* (*bdds* ! *i*) *ws* ! *a*) ∨ *bdd-lookup* (*bdd-binop bv-or bdd bdd′*) *ws* ! *a*) **by** (*simp add*: *bddh-binop bdd-lookup-binop bv-or-length*)
from 2 **have** ((∃ *i* < *length xs*. *xs* ! *i* ∧ *bdd-lookup* (*bdds* ! *i*) *ws* ! *a*) ∨ *bdd-lookup* (*bdd-binop bv-or bdd bdd′*) *ws* ! *a*) =
((∃ *i* < *length xs*. *xs* ! *i* ∧ *bdd-lookup* (*bdds* ! *i*) *ws* ! *a*) ∨ (*bdd-lookup bdd′ ws*) ! *a* ∨ (*bdd-lookup bdd ws*) ! *a*) **by** (*auto simp*: *bdd-lookup-binop bv-or-nth*)
also have ... = ((∃ *i* < *Suc* (*length xs*). (*True* # *xs*) ! *i* ∧ *bdd-lookup* ((*bdd′* # *bdds*) ! *i*) *ws* ! *a*) ∨ *bdd-lookup bdd ws* ! *a*)
(**is** ((∃ *i*. *?P i*) ∨ *?Q* ∨ *?R*) = ((∃ *i*. *?S i*) ∨ *?R*)) **proof**
assume (∃ *i*. *?P i*) ∨ *?Q* ∨ *?R* **thus** (∃ *i*. *?S i*) ∨ *?R* **by** (*elim disjE*) *auto*
next

```
    assume (∃ i. ?S i) ∨ ?R thus (∃ i. ?P i) ∨ ?Q ∨ ?R proof (elim disjE)
      assume ∃ i. ?S i
      then obtain i where I: ?S i ..
      { assume i = 0 with I have ?Q by simp }
      { assume i ≠ 0 then obtain j where i = Suc j by (cases i) simp+ with
I have ∃ 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 $\forall\, bd \in set\ (fst\ N).\ nfa\text{-}is\text{-}node\ N\ (bdd\text{-}lookup\ bd\ w)$ **by** (*simp add*: *bdd-all-bdd-lookup*)
}
with *assms* **have** $bdd\text{-}lookup\ (subsetbdd\ (fst\ N)\ q\ (nfa\text{-}emptybdd\ (length\ q)))\ ws\ !\ a = ((\exists\, i < length\ q.\ q\ !\ i \wedge bdd\text{-}lookup\ (fst\ N\ !\ i)\ ws\ !\ a) \vee bdd\text{-}lookup\ (nfa\text{-}emptybdd\ (length\ q))\ ws\ !\ a)$
by (*simp add*: *bdd-lookup-subsetbdd' nfa-is-node-def wf-nfa-def list-all-iff nfa-emptybdd-def*)
with *assms* **show** *?thesis* **by** (*auto simp*: *nfa-emptybdd-def nfa-is-node-def*)
qed

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

fun $nfa\text{-}accepting' :: bool\ list \Rightarrow bool\ list \Rightarrow bool$ **where**
$nfa\text{-}accepting'\ []\ bs = False$
$|\ nfa\text{-}accepting'\ as\ [] = False$
$|\ nfa\text{-}accepting'\ (a\ \#\ as)\ (b\ \#\ bs) = (a \wedge b \vee nfa\text{-}accepting'\ as\ bs)$
definition $nfa\text{-}accepting :: nfa \Rightarrow bool\ list \Rightarrow bool$ **where**
$nfa\text{-}accepting\ A = nfa\text{-}accepting'\ (snd\ A)$

lemma $nfa\text{-}accepting'\text{-}set\text{-}of\text{-}bv$: $nfa\text{-}accepting'\ l\ r = (set\text{-}of\text{-}bv\ l \cap set\text{-}of\text{-}bv\ r \neq \{\})$
proof −
have $nfa\text{-}accepting\text{-}help$: $\bigwedge as\ q.\ nfa\text{-}accepting'\ as\ q = (\exists\, i.\ i < length\ as \wedge i < length\ q \wedge as\ !\ i \wedge q\ !\ i)$
proof −
fix *as q*
show $nfa\text{-}accepting'\ as\ q = (\exists\, i < length\ as.\ i < length\ q \wedge as\ !\ i \wedge q\ !\ i)$
proof (*induct as q rule*: *nfa-accepting'.induct*)
case (3 *a as q qs*)
thus *?case* **proof** (*cases a∧q*)
case *False*
with 3 **have** $nfa\text{-}accepting'\ as\ qs = (\exists\, i < length\ as.\ i < length\ qs \wedge as\ !\ i \wedge qs\ !\ i)$ (**is** *?T* = -) **by** *simp*
also have $\ldots = (\exists j < length\ as.\ j < length\ qs \wedge (a\#as)\ !\ Suc\ j \wedge (q\#qs)\ !\ Suc\ j)$ **by** *simp*
also have $\ldots = (\exists j < length\ (a\#as).\ j < length\ (q\#qs) \wedge (a\#as)\ !\ j \wedge (q\#qs)\ !\ j)$ (**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** $\ldots = ((a \wedge q \wedge\ ?Q\ 0) \vee (\neg\ (a \wedge q) \wedge (\exists\, j.\ ?Q\ j)))$ **by** *auto*
also have $\ldots = ((a \wedge q \wedge (\exists\, j.\ ?Q\ j)) \vee (\neg (a \wedge q) \wedge (\exists\, j.\ ?Q\ j)))$ **by** *auto*
also have $\ldots = (\exists\, j.\ ?Q\ j)$ **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* $= (\exists\, i.\ i < length\ l \wedge i < length\ r \wedge l\ !\ i \wedge r\ !\ i)$ **by** *simp*
also have $\ldots = (\exists\, i.\ i \in set\text{-}of\text{-}bv\ l \wedge i \in set\text{-}of\text{-}bv\ r)$ **by** (*auto simp*: *set-of-bv-def*)
also have $\ldots = (set\text{-}of\text{-}bv\ l \cap set\text{-}of\text{-}bv\ r \neq \{\})$ **by** *auto*
finally show *?thesis* **.**
qed

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

definition
nfa-startnode :: *nfa* $\Rightarrow$ *bool list* **where**
nfa-startnode A $= (replicate\ (length\ (fst\ A))\ False)[0{:=}True]$

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* ≡ *foldl* (*nfa-trans A*)
abbreviation *nfa-accepts A* ≡ *accepts* (*nfa-trans A*) (*nfa-accepting A*) (*nfa-startnode A*)
abbreviation *nfa-reach A* ≡ *reach* (*nfa-trans A*)

lemma *nfa-startnode-is-node*: *wf-nfa A n* ⟹ *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* ⇒ *dfa* **where**
negate-dfa = (λ(*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* = (¬ *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* ⇒ *dfa* ⇒ *nat* × *nat* ⇒ (*nat* × *nat*) *list* **where**
prod-succs A B = (λ(*i*, *j*). *add-leaves* (*bdd-binop Pair* (*fst A* ! *i*) (*fst B* ! *j*)) [])

definition
prod-is-node A B = (λ(*i*, *j*). *dfa-is-node A i* ∧ *dfa-is-node B j*)

definition
prod-invariant :: *dfa* ⇒ *dfa* ⇒ *nat option list list* × (*nat* × *nat*) *list* ⇒ *bool* **where**
prod-invariant A B = (λ(*tab*, *ps*).
length tab = *length* (*fst A*) ∧ (∀ *tab*′∈*set tab*. *length tab*′ = *length* (*fst B*)))

definition
prod-ins = (λ(*i*, *j*). λ(*tab*, *ps*).
(*tab*[*i* := (*tab* ! *i*)[*j* := *Some* (*length ps*)]],
ps @ [(*i*, *j*)]))

definition
prod-memb :: *nat* × *nat* ⇒ *nat option list list* × (*nat* × *nat*) *list* ⇒ *bool* **where**
prod-memb = (λ(*i*, *j*). λ(*tab*, *ps*). *tab* ! *i* ! *j* ≠ *None*)

definition
prod-empt :: *dfa* ⇒ *dfa* ⇒ *nat option list list* × (*nat* × *nat*) *list* **where**
prod-empt A B = (*replicate* (*length* (*fst A*)) (*replicate* (*length* (*fst B*)) *None*), [])

definition
prod-dfs :: *dfa* ⇒ *dfa* ⇒ *nat* × *nat* ⇒ *nat option list list* × (*nat* × *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* ⇒ *bool* ⇒ *bool*) ⇒ *dfa* ⇒ *dfa* ⇒ *dfa* **where**
binop-dfa f A B =
(*let* (*tab*, *ps*) = *prod-dfs A B* (0, 0)
in
(*map* (λ(*i*, *j*). *bdd-binop* (λ*k l*. *the* (*tab* ! *k* ! *l*)) (*fst A* ! *i*) (*fst B* ! *j*)) *ps*,
map (λ(*i*, *j*). *f* (*snd A* ! *i*) (*snd B* ! *j*)) *ps*))

locale *prod-DFS* =
fixes *A B n*
assumes *well-formed*1: *wf-dfa A n*
and *well-formed*2: *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-formed*1 *well-formed*2)[]
apply (*simp add*: *prod-is-node-def prod-succs-def split-paired-all dfa-is-node-def wf-dfa-def*)
apply (*drule conjunct*1 [*OF conjunct*2])+
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*=[], *simplified*]])
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* ⟹ *prod-is-node A B y* ⟹
prod-memb y (*prod-dfs A B x*) = ((*x*, *y*) ∈ (*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* ∧ *dfa-is-node A i* ∧ *dfa-is-node B j*) =
(*k* < *length* (*snd* (*prod-dfs A B x*)) ∧ (*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* ∧ *dfa-is-node A i* ∧ *dfa-is-node B j*) =
(*k* < *length* (*snd* (*dfs* (*prod-empt A B*) [*x*])) ∧ (*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⟧ ⟹ fst (prod-dfs A B (i, j)) ! i ! j = Some 0
```

apply (*simp add*: *prod-dfs-def empt 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* < *length* (*snd* (*prod-dfs A B x*)) ∧ *snd* (*prod-dfs A B x*) ! *k* = (*i1*, *i2*)
by (*simp add*: *prod-dfs-bij* [*symmetric*])
moreover from *x j1 j2 j*
have *k* < *length* (*snd* (*prod-dfs A B x*)) ∧ *snd* (*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* (*dfa-trans A i bs*)
and *dfa-is-node B* (*dfa-trans B j bs*)
and *k′* < *length* (*snd* (*prod-dfs A B* (*s1*, *s2*)))
proof −
from *i well-formed1 bs* **have** *h-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-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*)) ∈ (*succsr* (*prod-succs A B*))*
using *i j s1 s2*

```
    by (simp add: prod-dfs-eq-rtrancl prod-is-node-def)
  moreover from h-tr1 h-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-s*1-*s*2, *symmetric*])
from *well-formed*1 *i* **have** *bddh-tr*1: *bddh n* (*fst A* ! *i*)
and *less-tr*1: *bdd-all* (*dfa-is-node A*) (*fst A* ! *i*) **by** (*simp add*: *wf-dfa-def list-all-iff dfa-is-node-def*)+
from *well-formed*2 *j* **have** *bddh-tr*2: *bddh n* (*fst B* ! *j*)
and *less-tr*2: *bdd-all* (*dfa-is-node B*) (*fst B* ! *j*) **by** (*simp add*: *wf-dfa-def list-all-iff dfa-is-node-def*)+
from *bddh-tr*1 *bddh-tr*2 **have** 1: *bddh n* (*bdd-binop* (λ*k l*. *the* (*fst* (*prod-dfs A B* (0, 0)) ! *k* ! *l*)) (*fst A* ! *i*) (*fst B* ! *j*))
by (*simp add*: *bddh-binop*)
have ∀ *bs*. *length bs* = *n* ⟶ *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-formed*1 *well-formed*2 **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-tr*1 *bddh-tr*2 **have** 2: *bdd-all* (λ*q*. *q* < *length* (*snd* (*prod-dfs A B* (0, 0)))) (*bdd-binop* (λ*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* - λ*x*. *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* (λ*q*. *q* < *length ?tr*)) *?tr* **by** (*auto simp*: *split-paired-all list-all-iff*)
from *well-formed*1 *well-formed*2 **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-s*1-*s*2 **have** (*fst* (*prod-dfs A B* (0,0)) ! 0 ! 0 = *Some* 0 ∧ *dfa-is-node A* 0 ∧ *dfa-is-node B* 0) =
(0 < *length* (*snd* (*prod-dfs A B* (0,0))) ∧ *snd* (*prod-dfs A B* (0,0)) ! 0 = (0,0)) **by** (*rule prod-dfs-bij*)
with 3 *well-formed*1 *well-formed*2 **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 (∃ *m*. *dfa-reach* (*binop-dfa f A B*) 0 *bss m* ∧
fst (*prod-dfs A B* (0, 0)) ! s_1 ! s_2 = *Some m* ∧
dfa-is-node A s_1 ∧ *dfa-is-node B* s_2) =
(*dfa-reach A* 0 *bss* s_1 ∧ *dfa-reach B* 0 *bss* s_2)
proof −
let *?tr* = *map* (λ(*i*, *j*).
bdd-binop (λ*k l*. *the* (*fst* (*prod-dfs A B* (0,0)) ! *k* ! *l*)) (*fst A* ! *i*) (*fst B* ! *j*))
(*snd* (*prod-dfs A B* (0,0)))
have *T*: *?tr* = *fst* (*binop-dfa f A B*) **by** (*simp add*: *binop-dfa-def split-beta*)
from *well-formed*1 *well-formed*2 **have** *is-node-s*1-*s*2: *prod-is-node A B* (0, 0) **by** (*simp add*: *prod-is-node-def wf-dfa-def dfa-is-node-def*)
from *well-formed*1 *well-formed*2 **have** *s*1: *dfa-is-node A* 0 **and** *s*2: *dfa-is-node B* 0 **by** (*simp add*: *dfa-is-node-def wf-dfa-def*)+
from *s*1 *s*2 **have** *start*: *fst* (*prod-dfs A B* (0,0)) ! 0 ! 0 = *Some* 0
by (*rule prod-dfs-start*)
show (∃ *m*. *dfa-reach* (*binop-dfa f A B*) 0 *bss m* ∧
fst (*prod-dfs A B* (0, 0)) ! s_1 ! s_2 = *Some m* ∧
dfa-is-node A s_1 ∧ *dfa-is-node B* s_2) =
(*dfa-reach A* 0 *bss* s_1 ∧ *dfa-reach B* 0 *bss* s_2)
(**is** (∃ *m*. *?lhs*1 *m* ∧ *?lhs*2 *m* ∧ *?lhs*3 ∧ *?lhs*4) = *?rhs*
is *?lhs* = -)
proof
assume ∃ *m*. *?lhs*1 *m* ∧ *?lhs*2 *m* ∧ *?lhs*3 ∧ *?lhs*4
then obtain *m* **where** *lhs*: *?lhs*1 *m* *?lhs*2 *m* *?lhs*3 *?lhs*4 **by** *auto*
from *lhs bss* **show** *?rhs*
proof (*induct arbitrary*: s_1 s_2)
case *Nil*
from *is-node-s*1-*s*2
*s*1 *s*2 ‹*dfa-is-node A* s_1› ‹*dfa-is-node B* s_2›
have (0, 0) = (s_1, s_2)
using *start* ‹*fst* (*prod-dfs A B* (0,0)) ! s_1 ! s_2 = *Some* 0›
by (*rule prod-dfs-inj*)
moreover have *dfa-reach A* 0 [] 0 **by** (*rule reach-nil*)
moreover have *dfa-reach B* 0 [] 0 **by** (*rule reach-nil*)
ultimately show *?case* **by** *simp*
next
case (*snoc j bss bs* s_1 s_2)
then have *length bs* = *n* **by** (*simp add*: *is-alph-def*)
moreover from *binop-wf-dfa* **have** *dfa-is-node* (*binop-dfa f A B*) 0 **by** (*simp add*: *dfa-is-node-def wf-dfa-def*)
with *snoc binop-wf-dfa* [*of f*] **have** *dfa-is-node* (*binop-dfa f A B*) *j* **by** (*simp add*: *dfa-reach-is-node*)
then have *j*: *j* < *length* (*snd* (*prod-dfs A B* (0,0))) **by** (*simp add*: *binop-dfa-def dfa-is-node-def split-beta*)
with *prod-dfs-bij* [*OF is-node-s*1-*s*2,

```
      of fst (snd (prod-dfs A B (0,0)) ! j) snd (snd (prod-dfs A B (0,0)) ! j)]
    have j-tr1: dfa-is-node A (fst (snd (prod-dfs A B (0,0)) ! j))
      and j-tr2: dfa-is-node B (snd (snd (prod-dfs A B (0,0)) ! j))
      and Some-j: fst (prod-dfs A B (0,0)) ! fst (snd (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 ∧ 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 s2' where reach-s2': dfa-reach B 0 bss s2' and s2': s2 = dfa-trans B
s2' 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 ! s2' = Some m
      and j: dfa-is-node A j and s2'': dfa-is-node B s2'
      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 s2''
    have m': (m < length (snd (prod-dfs A B (0,0))) ∧
      snd (prod-dfs A B (0,0)) ! m = (j, s2'))
      by (simp add: prod-dfs-bij [symmetric])
    with j s2'' 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 s2' 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 s2' bs)
      using j s2'' 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 *snd* (*binop-dfa f A B*) ! *dfa-steps* (*binop-dfa f A B*) 0 *bs* = *f* (*snd A* ! *dfa-steps A* 0 *bs*) (*snd B* ! *dfa-steps B* 0 *bs*)
(**is** *?as*3 ! *dfa-steps ?A* 0 *bs* = *?rhs*)
proof −
note 2 = *dfa-startnode-is-node*[*OF well-formed*1]
note 5 = *dfa-startnode-is-node*[*OF well-formed*2]
note *B* = *dfa-startnode-is-node*[*OF binop-wf-dfa*]
define *tab* **where** *tab* = *fst* (*prod-dfs A B* (0,0))
define *ps* **where** *ps* = *snd* (*prod-dfs A B* (0,0))
from *tab-def ps-def* **have** *prod*: *prod-dfs A B* (0,0) = (*tab*, *ps*) **by** *simp*
define *s*1 **where** *s*1 = *dfa-steps A* 0 *bs*
define *s*2 **where** *s*2 = *dfa-steps B* 0 *bs*
with *s*1-*def* **have** *dfa-reach A* 0 *bs s*1 **and** *dfa-reach B* 0 *bs s*2 **by** (*simp add*: *reach-def*)+
with *X* **have** ∃ *m*. *dfa-reach ?A* 0 *bs m* ∧ *fst* (*prod-dfs A B* (0, 0)) ! *s*1 ! *s*2 = *Some m* ∧ *dfa-is-node A s*1 ∧ *dfa-is-node B s*2
by (*simp add*: *binop-dfa-reachable*)
with *tab-def* **have** ∃ *m*. *dfa-reach ?A* 0 *bs m* ∧ *tab* ! *s*1 ! *s*2 = *Some m* ∧ *dfa-is-node A s*1 ∧ *dfa-is-node B s*2 **by** *simp*
then obtain *m* **where** *R*: *dfa-reach ?A* 0 *bs m* **and** *M*: *tab* ! *s*1 ! *s*2 = *Some m* **and** *s*1: *dfa-is-node A s*1 **and** *s*2: *dfa-is-node B s*2 **by** *blast*
hence *M'*: *m* = *dfa-steps ?A* 0 *bs* **by** (*simp add*: *reach-def*)
from *B X R binop-wf-dfa* [*of f*] **have** *mL*: *dfa-is-node ?A m* **by** (*simp add*: *dfa-reach-is-node*)
from 2 5 *M s*1 *s*2 **have** *bij*: *m* < *length* (*snd* (*prod-dfs A B* (0, 0))) ∧ *snd* (*prod-dfs A B* (0, 0)) ! *m* = (*s*1, *s*2) **unfolding** *tab-def*
by (*simp add*: *prod-dfs-bij*[*symmetric*] *prod-is-node-def*)
with *mL* **have** *snd* (*binop-dfa f A B*) ! *m* = *f* (*snd A* ! *s*1) (*snd B* ! *s*2)
by (*simp add*: *binop-dfa-def split-beta dfa-is-node-def*)
with *M'* *s*1-*def s*2-*def*
show *snd ?A* ! *dfa-steps ?A* 0 *bs* = *f* (*snd A* ! *dfa-steps A* 0 *bs*) (*snd B* ! *dfa-steps B* 0 *bs*)
by *simp*
qed

end

lemma *binop-wf-dfa*:
assumes *A*: *wf-dfa A n* **and** *B*: *wf-dfa B n*
shows *wf-dfa* (*binop-dfa f A B*) *n*
proof −
from *A B*
interpret *prod-DFS A B n* **by** *unfold-locales*
show *?thesis* **by** (*rule binop-wf-dfa*)

qed

theorem *binop-dfa-accepts*:
assumes *A*: *wf-dfa A n*
and *B*: *wf-dfa B n*
and *X*: *list-all* (*is-alph n*) *bss*
shows *dfa-accepts* (*binop-dfa f A B*) *bss* = *f* (*dfa-accepts A bss*) (*dfa-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-dfa :: *dfa* ⇒ *dfa* ⇒ *dfa* **where**
and-dfa = *binop-dfa* (∧)

lemma *and-wf-dfa*:
assumes *wf-dfa M n*
and *wf-dfa N n*
shows *wf-dfa* (*and-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-accepts* (*and-dfa M N*) *bs* = (*dfa-accepts M bs* ∧ *dfa-accepts N bs*)
using *assms* **by** (*simp add*: *binop-dfa-accepts and-dfa-def*)

definition
or-dfa :: *dfa* ⇒ *dfa* ⇒ *dfa* **where**
or-dfa = *binop-dfa* (∨)

lemma *or-wf-dfa*:
assumes *wf-dfa M n* **and** *wf-dfa N n*
shows *wf-dfa* (*or-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-accepts* (*or-dfa M N*) *bs* = (*dfa-accepts M bs* ∨ *dfa-accepts N bs*)
using *assms* **by** (*simp add*: *binop-dfa-accepts or-dfa-def*)

definition
imp-dfa :: *dfa* ⇒ *dfa* ⇒ *dfa* **where**

imp-dfa = *binop-dfa* (⟶)

lemma *imp-wf-dfa*:
assumes *wf-dfa M n* **and** *wf-dfa N n*
shows *wf-dfa* (*imp-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* ⟶ *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* ⇒ *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 ⋀*a*. *dfa-is-node M a* ⟹ *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 ⋀*bdd*. *bdd-all* (*dfa-is-node M*) *bdd* ⟹ *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*: ⟦*q* < *n*; (*replicate n False*)[*q*:=*True*] = (*replicate n False*)[*p*:=*True*]⟧ ⟹ (*q* = *p*) (**is** ⟦- ;*?lhs* = *?rhs*⟧ ⟹ -)
proof −
assume *q*: *q* < *n* **and** *r*: *?lhs* = *?rhs*
{ assume *p* ≠ *q*
with *q* **have** *?lhs* ! *q* = *True* **by** *simp*
moreover from ‹*p* ≠ *q*› *q* **have** *?rhs* ! *q* = *False* **by** *simp*
ultimately have *?lhs* ≠ *?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 $\exists p.$ *dfa-reach M q bss p* $\wedge$ $n2 = (replicate\ (length\ (fst\ M))\ False)[p{:=}True]$
proof −
from *R V X N Q* **show** *?thesis* **proof** *induct*
case *Nil*
hence *dfa-reach M q* [] *q* **by** (*simp add*: *reach-nil*)
with *Nil* **show** *?case* **by** *auto*
next
case (*snoc j bss bs*)
hence $N1$: *nfa-is-node (nfa-of-dfa M) n1* **by** (*simp add*: *nfa-is-node-def nfa-of-dfa-def split-beta*)
from *snoc* **have** $V2$: *wf-nfa (nfa-of-dfa M) l* **by** (*simp add*: *dfa2wf-nfa*)
from *snoc* **have** $\exists p.$ *dfa-reach M q bss p* $\wedge$ $j = (replicate\ (length\ (fst\ M))\ False)[p := True]$ **by** *simp*
then obtain *p* **where** *PR*: *dfa-reach M q bss p* **and** J: $j = (replicate\ (length\ (fst\ M))\ False)[p{:=}True]$ **by** *blast*
hence *JL*: *nfa-is-node (nfa-of-dfa M) j* **by** (*simp add*: *nfa-is-node-def nfa-of-dfa-def split-beta*)
from *snoc PR* **have** *PL*: *dfa-is-node M p* **by** (*simp add*: *dfa-reach-is-node*)
with *snoc JL* **have** *PL'*: *p < length j* **by** (*simp add*: *nfa-is-node-def dfa-is-node-def nfa-of-dfa-def split-beta*)
define *m* **where** *m = dfa-trans M p bs*
with *snoc PR* **have** *MR*: *dfa-reach M q (bss @ [bs]) m* **by** (*simp add*: *reach-snoc*)
with *snoc* **have** *mL*: *dfa-is-node M m* **by** (*simp add*: *dfa-reach-is-node*)
from $V2$ *JL snoc* **have** *nfa-is-node (nfa-of-dfa M) (nfa-trans (nfa-of-dfa M) j bs)* **by** (*simp add*: *nfa-trans-is-node*)
hence *L*: *length (nfa-trans (nfa-of-dfa M) j bs) = length (fst M)* **by** (*simp add*: *nfa-is-node-def nfa-of-dfa-def split-beta*)

have *nfa-trans (nfa-of-dfa M) j bs* $= (replicate\ (length\ (fst\ M))\ False)[m := True]$ (**is** *?lhs = ?rhs*)
proof (*simp add*: *list-eq-iff-nth-eq L, intro strip*)
fix *i* **assume** *H*: *i < length (fst M)*
show *nfa-trans (nfa-of-dfa M) j bs* ! *i* $= (replicate\ (length\ (fst\ M))\ False)[m := True]$! *i* (**is** *?lhs = ?rhs*)
proof
assume *lhs*: *?lhs*
from $V2$ *snoc* **have** *wf-nfa (nfa-of-dfa M) (length bs)* **by** (*simp add*: *is-alph-def*) **moreover**
note *JL* **moreover**
from *H* **have** *IL*: *i < length (fst (nfa-of-dfa M))* **by** (*simp add*: *nfa-of-dfa-def split-beta*) **moreover**
from ‹*?lhs*› **have** *bdd-lookup (subsetbdd (fst (nfa-of-dfa M)) j (nfa-emptybdd (length j))) bs* ! *i* **by** (*simp add*: *nfa-trans-def*)
ultimately have $\exists x <$ *length j.* j ! $x \wedge$ *bdd-lookup (fst (nfa-of-dfa M)* ! x)

bs ! *i* **by** (*simp add*: *bdd-lookup-subsetbdd*)
then obtain *x* **where** *xl*: *x* < *length j* **and** *xj*: *j* ! *x* **and** *xs*: *bdd-lookup* (*fst* (*nfa-of-dfa M*) ! *x*) *bs* ! *i* **by** *blast*
with *snoc J PL′* **have** *x* = *p* **by** (*cases p* = *x*) *simp*+
with *xs PL snoc*(3,4) *m-def* **show** (*replicate* (*length* (*fst M*)) *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) *V*2 **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 *N*1: *n*1 = (*replicate* (*length* (*fst M*)) *False*)[*q*:=*True*]
and *N*2: *n*2 = (*replicate* (*length* (*fst M*)) *False*)[*p*:=*True*]
and *Q*: *dfa-is-node M q*
shows *nfa-reach* (*nfa-of-dfa M*) *n*1 *bss n*2 = *dfa-reach M q bss p*
proof
assume *nfa-reach* (*nfa-of-dfa M*) *n*1 *bss n*2
with *assms* **have** ∃ *p*. *dfa-reach M q bss p* ∧ *n*2 = (*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** *N*2′: *n*2 = (*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 *N*2 *N*2′ **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** $n2' = nfa\text{-}steps\ (nfa\text{-}of\text{-}dfa\ M)\ n1\ bss$
hence R': $nfa\text{-}reach\ (nfa\text{-}of\text{-}dfa\ M)\ n1\ bss\ n2'$ **by** $(simp\ add{:}\ reach\text{-}def)$
with $assms$ **have** $\exists p.\ dfa\text{-}reach\ M\ q\ bss\ p \wedge n2' = (replicate\ (length\ (fst\ M))\ False)[p := True]$ **by** $(simp\ add{:}\ nfa\text{-}of\text{-}dfa\text{-}reach')$
then obtain p' **where** R: $dfa\text{-}reach\ M\ q\ bss\ p'$ **and** $N2'$: $n2' = (replicate\ (length\ (fst\ M))\ False)[p' := True]$ **by** $blast$
with H **have** $p = p'$ **by** $(simp\ add{:}\ reach\text{-}inj)$
with $N2'\ N2$ **have** $n2 = n2'$ **by** $simp$
with R' **show** $nfa\text{-}reach\ (nfa\text{-}of\text{-}dfa\ M)\ n1\ bss\ n2$ **by** $simp$
qed

lemma $nfa\text{-}accepting\text{-}replicate$:
assumes $q < length\ (fst\ N)$
and $length\ (snd\ N) = length\ (fst\ N)$
shows $nfa\text{-}accepting\ N\ ((replicate\ (length\ (fst\ N))\ False)[q{:=}True]) = snd\ N\ !\ q$
proof −
from $assms$ **have** $set\text{-}of\text{-}bv\ ((replicate\ (length\ (fst\ N))\ False)[q{:=}True]) = \{q\}$
proof $(auto\ simp{:}\ set\text{-}of\text{-}bv\text{-}def)$
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 $nfa\text{-}accepting\ N\ ((replicate\ (length\ (fst\ N))\ False)[q{:=}True]) = (set\text{-}of\text{-}bv\ (snd\ N) \cap \{q\} \neq \{\})$
by $(simp\ add{:}\ nfa\text{-}accepting\text{-}set\text{-}of\text{-}bv)$
also have $\ldots = (q \in set\text{-}of\text{-}bv\ (snd\ N))$ **by** $auto$
also from $assms$ **have** $\ldots = snd\ N\ !\ q$ **by** $(auto\ simp{:}\ set\text{-}of\text{-}bv\text{-}def)$
finally show $?thesis$ **.**
qed

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

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* ⇒ *bool list* ⇒ *'a* ⇒ *'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* ≥ *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*) ∧ 0 ≤ *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* ∧ *bddh y ll* ∧ *y* ≤ *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* ≤ *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* ≤ *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 ∃ *v vs*. *vv* = *v* # *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 ∃ *v vs*. *vv* = *v* # *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* ⇒ *bool list* ⇒ *bool list list* **where**
subset-succs A qs = *add-leaves* (*subsetbdd* (*fst A*) *qs* (*nfa-emptybdd* (*length qs*)))
[]

definition
subset-invariant :: *nfa* ⇒ *nat option bdd* × *bool list list* ⇒ *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* ⇒ *nat option bdd* × *bool list list* ⇒ *bool* **where**
subset-memb qs = (λ(*bdd*, *qss*). *bdd-lookup bdd qs* ≠ *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* (λ*qs*. *bdd-map* (λ*qs*. *the* (*bdd-lookup bdd qs*)) (*subsetbdd* (*fst A*) *qs* (*nfa-emptybdd* (*length qs*)))) *qss*,
map (λ*qs*. *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 *H*1: *nfa-is-node A q*
and *H*2: *nfa-is-node A q*0
shows (*bdd-lookup* (*fst* (*subset-dfs A q*0)) *q* = *Some v*) = (*v* < *length* (*snd* (*subset-dfs A q*0)) ∧ (*snd* (*subset-dfs A q*0)) ! *v* = *q*)
proof −
from *assms* **have** *list-all* (*nfa-is-node A*) [*q*0] **by** *simp*
with *empt-invariant* **show** *?thesis* **using** *H*1 **unfolding** *subset-dfs-def*
proof (*induct arbitrary*: *v q rule*: *dfs-invariant*)
case (*step S x vv qq*)
obtain *bd*1 *l*1 **where** *S*: *S* = (*bd*1, *l*1) **by** (*cases S*) *blast*+
{ assume *x* ∈ *set l*1
hence *list-ex* (λ*l*. *l* = *x*) *l*1 **by** (*simp add*: *list-ex-iff*)
hence ∃ *i* < *length l*1. *l*1 ! *i* = *x* **by** (*simp add*: *list-ex-length*)
then obtain *i* **where** *i* < *length l*1 ∧ *l*1 ! *i* = *x* **by** *blast*
with *step S* **have** *bdd-lookup bd*1 *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 l*1. *l*1 ! *i* ≠ *x* **by** *auto*
obtain *bd*2 *l*2 **where** *S*2: *subset-ins x S* = (*bd*2, *l*2) **by** (*cases subset-ins x S*) *blast*+
with *S* **have** *SS*: *bd*2 = *bddinsert bd*1 *x* (*Some* (*length l*1)) *l*2 = *l*1 @ [*x*] **by** (*simp add*: *subset-ins-def*)+
from *step S H*1 **have** *bdd-lookup* (*bddinsert bd*1 *x* (*Some* (*length l*1))) *qq* = (*if x* = *qq then Some* (*length l*1) *else bdd-lookup bd*1 *qq*)
by (*simp add*: *bdd-lookup-bddinsert subset-invariant-def nfa-is-node-def*)
with *SS* **have** (*bdd-lookup bd*2 *qq* = *Some vv*) = (*if x* = *qq then length l*1 = *vv else bdd-lookup bd*1 *qq* = *Some vv*) **by** *simp*
also have ... = (*x* = *qq* ∧ *length l*1 = *vv* ∨ *x* ≠ *qq* ∧ *bdd-lookup bd*1 *qq* = *Some vv*) **by** *auto*
also have ... = (*vv* < *length l*2 ∧ *l*2 ! *vv* = *qq*) **proof** (*cases x* = *qq*)
case *True*
hence (*x* = *qq* ∧ *length l*1 = *vv* ∨ *x* ≠ *qq* ∧ *bdd-lookup bd*1 *qq* = *Some vv*) = (*x* = *qq* ∧ *length l*1 = *vv*) **by** *simp*
also have ... = (*vv* < *length l*2 ∧ *l*2 ! *vv* = *qq*) **proof**
assume *H*: *vv* < *length l*2 ∧ *l*2 ! *vv* = *qq*
show *x* = *qq* ∧ *length l*1 = *vv* **proof** (*cases vv* = *length l*1)

```
        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
      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 q*0
shows *list-all* (*nfa-is-node A*) (*snd* (*subset-dfs A q*0))
proof −
from *assms* **have** *list-all* (*nfa-is-node A*) [*q*0] **by** *simp*
with *empt-invariant* **show** *?thesis* **unfolding** *subset-dfs-def*
proof (*induct rule*: *dfs-invariant*)
case *base* **thus** *?case* **by** (*simp add*: *subset-empt-def*)
next
case (*step S x*) **thus** *?case* **by** (*simp add*: *subset-ins-def split-beta*)
qed
qed

lemma *det-wf-nfa*:
shows *wf-dfa* (*det-nfa A*) *n*
proof −
obtain *bt ls* **where** *BT*: *subset-dfs A* (*nfa-startnode A*) = (*bt*, *ls*) **by** (*cases subset-dfs A* (*nfa-startnode A*)) *auto*
note *Q* = *nfa-startnode-is-node*[*OF well-formed*]
from *Q* **have** *N*:*list-all* (*nfa-is-node A*) (*snd* (*subset-dfs A* (*nfa-startnode A*)))
by (*simp add*: *subset-dfs-is-node*)
with *BT* **have** *L*: *list-all* (*nfa-is-node A*) *ls* **by** *simp*
have *D*: *det-nfa A* = (*map* (λ*q*. *bdd-map* (λ*q*. *the* (*bdd-lookup bt q*)) (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*)))) *ls*, *map* (λ*q*. *nfa-accepting A q*) *ls*)
(**is** - = (*?bdt*, *?atbl*)) **unfolding** *det-nfa-def* **by** (*simp add*: *BT*)
from *well-formed L* **have** *list-all* (λ*q*. *bddh n* (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*)))) *ls*
by (*induct ls*) (*simp add*: *bddh-subsetbdd wf-nfa-def nfa-emptybdd-def*)+
hence *list-all* (λ*q*. *bddh n* (*bdd-map* (λ*q*. *the* (*bdd-lookup bt q*)) (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*))))) *ls*
by (*simp add*: *bddh-bdd-map*)
hence *A*: *list-all* (*bddh n*) *?bdt* **by** (*simp add*: *list-all-iff*)
{
fix *q* **assume** ∃ *i* < *length ls*. *ls* ! *i* = *q*
then obtain *i* **where** *len-i*: *i* < *length ls* **and** *i*: *q* = *ls* ! *i* **by** *blast*
from *len-i i L* **have** *Q'*: *nfa-is-node A q* **by** (*simp add*: *list-all-iff*)
then have (*bdd-lookup* (*fst* (*subset-dfs A* (*nfa-startnode A*))) *q* = *Some i*) =
(*i* < *length* (*snd* (*subset-dfs A* (*nfa-startnode A*))) ∧ *snd* (*subset-dfs A* (*nfa-startnode A*)) ! *i* = *q*)
using *Q*
by (*rule subset-dfs-bij*)
with *BT len-i i* **have** *bdd-lookup bt q* = *Some i* **by** *simp*
with *BT* **have** *subset-memb q* (*subset-dfs A* (*nfa-startnode A*)) **by** (*simp add*: *subset-memb-def*)
with *Q' Q* **have** *TR*: (*nfa-startnode A*,*q*) ∈ (*succsr* (*subset-succs A*))*
by (*simp add*: *dfs-eq-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-def rtrancl-into-rtrancl*)
from Q' **have** *list-all* (*nfa-is-node A*) (*subset-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* $\neq$ *None* **by** (*simp add*: *subset-memb-def*)
with BT **obtain** j **where** j: *bdd-lookup* (*fst* (*subset-dfs A* (*nfa-startnode A*))) *p* = *Some j* **by** (*cases bdd-lookup bt p*) *simp+*
from 4 *Q j* **have** *j* < *length* (*snd* (*subset-dfs A* (*nfa-startnode A*))) ∧ (*snd* (*subset-dfs A* (*nfa-startnode A*))) ! *j* = *p*
by (*auto simp add*: *subset-dfs-bij*)
with *j BT* 4 **have** ∃*j*. *bdd-lookup bt p* = *Some j* ∧ *j* < *length ls* **by** *auto*
}
hence ∀ *p* ∈ *set* (*subset-succs A q*). ∃*j*. *bdd-lookup bt p* = *Some j* ∧ *j* < *length ls* **by** *auto*
hence *list-all* (λ*p*. ∃*j*. *bdd-lookup bt p* = *Some j* ∧ *j* < *length ls*) (*add-leaves* (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*))) []) **by** (*simp add*: *list-all-iff subset-succs-def*)
hence *bdd-all* (λ*p*. ∃*j*. *bdd-lookup bt p* = *Some j* ∧ *j* < *length ls*) (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*))) **by** (*simp add*: *add-leaves-bdd-all-eq*)
hence *bdd-all* (λ*l*. *l* < *length ls*) (*bdd-map* (λ*q*. *the* (*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 ∀ *x* ∈ *set ls*. *bdd-all* (λ*l*. *l* < *length ls*) (*bdd-map* (λ*q*. *the* (*bdd-lookup bt q*)) (*subsetbdd* (*fst A*) *x* (*nfa-emptybdd* (*length x*)))) **by** (*simp add*: *in-set-conv-nth*)
hence *list-all* (λ*x*. *bdd-all* (λ*l*. *l* < *length ls*) (*bdd-map* (λ*q*. *the* (*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* (λ*l*. *l* < *length ls*)) (*map* (λ*x*. *bdd-map* (λ*q*. *the* (*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 < *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 (∃ *bss*. *nfa-reach A i bss j* ∧ *list-all* (*is-alph n*) *bss*) = ((*i*, *j*) ∈ (*succsr* (*subset-succs A*))*)
proof
assume ∃ *bss*. *nfa-reach A i bss j* ∧ *list-all* (*is-alph n*) *bss*
then obtain *bss* **where** *BS*: *nfa-reach A i bss j list-all* (*is-alph n*) *bss* **by** *blast*

```
  show (i,j) ∈ (succsr (subset-succs A))*
  proof −
    from BS show (i,j) ∈ (succsr (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 ∃ bss. nfa-reach A i bss j ∧ 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 ∃ bs. z = bdd-lookup (subsetbdd (fst A) y (nfa-emptybdd (length
y))) bs ∧ 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)
moreover
    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 q*0)
proof −
from *assms well-formed* **have** Q: *nfa-is-node A q* **by** (*simp add*: *nfa-reach-is-node*)
from *R X* **have** ∃ *bs*. *nfa-reach A q*0 *bs q* ∧ *list-all* (*is-alph n*) *bs* **by** *auto*
with Q0 **have** (*q*0,*q*) ∈ (*succsr* (*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* :: *nat option bdd* **and** *ls* :: *bool list list*
assumes *subset-dfs A* (*nfa-startnode A*) = (*bd*, *ls*) (**is** *?subset-dfs* = -)
and ∃ *bs*. *nfa-reach A* (*nfa-startnode A*) *bs q*1 ∧ *list-all* (*is-alph n*) *bs*
and *q*1 = *ls* ! *i* **and** *q*2 = *ls* ! *j* **and** *i* < *length ls* **and** *j* < *length ls*
and *list-all* (*is-alph n*) *bss*
shows *nfa-reach A q*1 *bss q*2 = (*dfa-reach* (*det-nfa A*) *i bss j* ∧ *nfa-is-node A q*2)
(**is** - = (*dfa-reach ?M i bss j* ∧ -))
proof
assume *nfa-reach A q*1 *bss q*2
from *this assms* **show** *dfa-reach ?M i bss j* ∧ *nfa-is-node A q*2
proof (*induct arbitrary*: *j*)
case (*Nil j*)
with *well-formed* **have** Q0: *nfa-is-node A* (*nfa-startnode A*) **by** (*simp add*: *nfa-startnode-is-node*)
from *Nil* **obtain** *bs* **where** *nfa-reach A* (*nfa-startnode A*) *bs q*1 **and** *list-all* (*is-alph n*) *bs* **by** *blast*
with *well-formed Q*0 *Nil* **have** Q1: *nfa-is-node A q*1 **by** (*simp add*: *nfa-reach-is-node*)
with Q0 **have** ⋀*v*. (*bdd-lookup* (*fst* (*?subset-dfs*)) *q*1 = *Some v*) = (*v* < *length* (*snd* (*?subset-dfs*)) ∧ *snd* (*?subset-dfs*) ! *v* = *q*1)
by (*simp add*: *subset-dfs-bij*)
with *Nil*(1) **have** 1: ⋀*v*. (*bdd-lookup bd q*1 = *Some v*) = (*v* < *length ls* ∧ *ls* ! *v* = *q*1) **by** *simp*
from *Nil* 1 **have** *bdd-lookup bd q*1 = *Some i* **by** *simp*
moreover from *Nil* 1 **have** *bdd-lookup bd q*1 = *Some j* **by** *simp*
ultimately have *i* = *j* **by** *simp*
have *dfa-reach ?M i* [] *i* **by** (*simp add*: *reach-nil*)
with ‹*i*=*j*› Q1 **show** *?case* **by** *simp*
next
case (*snoc p bss bs j*)
note *S-len* = *nfa-startnode-is-node*[*OF well-formed*]
from *snoc* **obtain** *bss′* **where** *BSS′*:*nfa-reach A* (*nfa-startnode A*) *bss′ q*1 **and** *BSS′L*: *list-all* (*is-alph n*) *bss′* **by** *blast*
with *well-formed S-len* **have** *Q-len*: *nfa-is-node A q*1 **by** (*simp add*: *nfa-reach-is-node*)
with *well-formed snoc* **have** *P-len*: *nfa-is-node A p* **by** (*simp add*: *nfa-reach-is-node*)
from *BSS′ snoc* **have** *nfa-reach A* (*nfa-startnode A*) (*bss′* @ *bss*) *p* **by** (*simp add*: *reach-trans*) **moreover**
note *S-len* **moreover**
from *snoc BSS′L* **have** *list-all* (*is-alph n*) (*bss′* @ *bss*) **by** *simp*

ultimately have *subset-memb p ?subset-dfs* **by** (*rule nfa-reach-subset-memb*)
hence *bdd-lookup* (*fst ?subset-dfs*) *p* ≠ *None* **by** (*simp add*: *subset-memb-def split-beta*)
then obtain *v* **where** *P*: *bdd-lookup* (*fst ?subset-dfs*) *p* = *Some v* **by** (*cases bdd-lookup* (*fst ?subset-dfs*) *p*) *simp+*
with *P-len S-len* **have** *v* < *length* (*snd* (*?subset-dfs*)) ∧ *snd ?subset-dfs* ! *v* = *p* **by** (*simp add*: *subset-dfs-bij*)
with *snoc* **have** *V*: *v* < *length ls* ∧ *ls* ! *v* = *p* **by** *simp*
with *snoc P-len* **have** *R*: *dfa-reach ?M i bss v* ∧ *nfa-is-node A p* **by** *simp*

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

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

from *snoc* **have** *BD*: *fst ?M* = *map* (λ*q*. *bdd-map* (λ*q*. *the* (*bdd-lookup bd q*)) (*subsetbdd* (*fst A*) *q* (*nfa-emptybdd* (*length q*)))) *ls*
by (*simp add*: *det-nfa-def*)
with *V* **have** *fst ?M* ! *v* = *bdd-map* (λ*q*. *the* (*bdd-lookup bd q*)) (*subsetbdd* (*fst A*) *p* (*nfa-emptybdd* (*length p*))) **by** *simp*
with *well-formed BS P-len* **have** *bdd-lookup* (*fst ?M* ! *v*) *bs* = *the* (*bdd-lookup bd* (*bdd-lookup* (*subsetbdd* (*fst A*) *p* (*nfa-emptybdd* (*length p*))) *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** ... = *j* **by** (*simp add*: *nfa-trans-def*)
finally have *JJ*: *bdd-lookup* (*fst ?M* ! *v*) *bs* = *j* **.**

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

from *snoc* **have** *L*: *length* (*fst ?M*) = *length ls* **by** (*simp add*: *det-nfa-def*)

with *snoc* **have** *dfa-is-node ?M i* **by** (*simp add*: *dfa-is-node-def*) **moreover**
note ‹*dfa-reach ?M i bss j*› **moreover**
from *snoc* **have** *wf-dfa ?M n* **by** (*simp add*: *det-wf-nfa*) **moreover**
from *snoc* **have** *list-all* (*is-alph n*) *bss* **by** *simp*
ultimately have *dfa-is-node ?M j* **by** (*simp add*: *dfa-reach-is-node*)
with *L* **have** *J-len*: *j* < *length ls* **by** (*simp add*: *dfa-is-node-def*)
from *Q*0 **have** *list-all* (*nfa-is-node A*) (*snd ?subset-dfs*) **by** (*rule subset-dfs-is-node*)
with *snoc J-len* **have** *J*: *nfa-is-node A* (*ls* ! *j*) **by** (*simp add*: *list-all-iff*)
moreover note *snoc*(4,5,6) *refl*[*of ls*!*j*] *snoc*(8) *J-len*
moreover from *snoc* **have** *list-all* (*is-alph n*) *bss* **by** *simp*
ultimately have *R*: *nfa-reach A q*1 *bss* (*ls* ! *j*) **by** (*rule snoc*(2))
from *snoc* **obtain** *bs′* **where** *R′*: *nfa-reach A* (*nfa-startnode A*) *bs′ q*1 **and** *BS′*: *list-all* (*is-alph n*) *bs′* **by** *blast*
with *R* **have** *lsj*: *nfa-reach A* (*nfa-startnode A*) (*bs′* @ *bss*) (*ls* ! *j*) **by** (*simp add*: *reach-trans*)
hence *nfa-reach A* (*nfa-startnode A*) ((*bs′* @ *bss*) @ [*bs*]) *qq* **unfolding** *qq-def* **by** (*rule reach-snoc*)
with *well-formed snoc*(10) *Q*0 *BS′* **have** *M*: *subset-memb qq ?subset-dfs* **and** *QQ-len*: *nfa-is-node A qq* **by** (*simp add*: *nfa-reach-subset-memb nfa-reach-is-node*)+
with *snoc*(4) **have** *QQ*: *bdd-lookup bd qq* ≠ *None* **by** (*simp add*: *subset-memb-def*)

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

with *snoc*(4) **have** *bdd-lookup* (*fst ?subset-dfs*) *qq* = *Some v* **by** *simp*
with *QQ-len Q*0 **have** *v* < *length* (*snd ?subset-dfs*) ∧ (*snd ?subset-dfs*) ! *v* = *qq* **by** (*simp add*: *subset-dfs-bij*)
with *snoc v-def* **have** *Q*2: *qq* = *q*2 **by** (*simp add*: *dfa-trans-def*)

with *R qq-def* **show** *nfa-reach A q*1 (*bss* @ [*bs*]) *q*2 **by** (*simp add*: *reach-snoc*)
qed (*simp add*: *reach-nil*)
qed

lemma *det-nfa-reach*:
fixes *bd* :: *nat option bdd* **and** *ls* :: *bool list list*
assumes *S*: *subset-dfs A* (*nfa-startnode A*) = (*bd*, *ls*) (**is** *?subset-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: $\exists b.\ nfa\text{-}reach\ A\ (nfa\text{-}startnode\ A)\ b\ (nfa\text{-}startnode\ A) \wedge list\text{-}all\ (is\text{-}alph\ n)\ b$ **by** $auto$
from SL **have** $bdd\text{-}lookup\ (fst\ ?subset\text{-}dfs)\ (nfa\text{-}startnode\ A) = Some\ 0$ **by** $(simp\ add: subset\text{-}dfs\text{-}start)$
with SL **have** $0 < length\ (snd\ ?subset\text{-}dfs) \wedge snd\ ?subset\text{-}dfs\ !\ 0 = nfa\text{-}startnode\ A$ **by** $(simp\ add: subset\text{-}dfs\text{-}bij)$
with S **have** 2: $0 < length\ ls \wedge ls\ !\ 0 = nfa\text{-}startnode\ A$ **by** $simp$
from $S\ 1\ Q1\ J\ 2\ X$ **have** T: $nfa\text{-}reach\ A\ (nfa\text{-}startnode\ A)\ bss\ q1 = (dfa\text{-}reach\ (det\text{-}nfa\ A)\ 0\ bss\ j \wedge nfa\text{-}is\text{-}node\ A\ q1)$
by $(simp\ only: det\text{-}nfa\text{-}reach')$
from SL **have** $list\text{-}all\ (nfa\text{-}is\text{-}node\ A)\ (snd\ ?subset\text{-}dfs)$ **by** $(simp\ add: subset\text{-}dfs\text{-}is\text{-}node)$
with $Q1\ J\ S$ **have** $nfa\text{-}is\text{-}node\ A\ q1$ **by** $(simp\ add: list\text{-}all\text{-}iff)$
with T **show** $?thesis$ **by** $simp$
qed

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

finally show *dfa-accepts* (*det-nfa A*) *w* = *nfa-accepts A 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* ⇒ *bool list bdd* ⇒ *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* ≤ *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*])+

definition
quantify-nfa :: *nat* ⇒ *nfa* ⇒ *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*)+
from 1 *assms* **have** 3: *list-all* (*bddh n*) (*fst* (*quantify-nfa v N*)) **by** (*simp add*: *quantify-nfa-def bddh-quantify-bdd list-all-iff split-beta*)
from 2 **have** *list-all* (*bdd-all* (*nfa-is-node N*)) (*fst* (*quantify-nfa v N*)) **by** (*simp add*: *quantify-bdd-is-node list-all-iff split-beta quantify-nfa-def*)
hence *list-all* (*bdd-all* (*nfa-is-node* (*quantify-nfa v N*))) (*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 *insertl* :: *nat* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* **where**
insertl i a [] = [*a*]
| *insertl* 0 *a bs* = *a* # *bs*
| *insertl* (*Suc i*) *a* (*b* # *bs*) = *b* # (*insertl i a bs*)

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

lemma *insertl*-0-*eq*: *insertl* 0 *x xs* = *x* # *xs*
by (*cases xs*) *simp-all*

lemma *bdd-lookup-quantify-bdd-set-of-bv*:
assumes *length w* = *n*
and *bddh* (*Suc n*) *bdd*
and *bdd-all* (*nfa-is-node N*) *bdd*
and $v \leq n$
shows *set-of-bv* (*bdd-lookup* (*quantify-bdd v bdd*) *w*) = (⋃ *b*. *set-of-bv* (*bdd-lookup 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*: *nfa-is-node N* (*bdd-lookup l w*) *nfa-is-node N* (*bdd-lookup r w*) **by** (*simp add*: *bdd-all-bdd-lookup*)+
have *set-of-bv* (*bdd-lookup* (*quantify-bdd* 0 (*Branch l r*)) *w*) = *set-of-bv* (*bdd-lookup* (*bdd-binop bv-or l r*) *w*) **by** *simp*
also from 2 **have** ... = *set-of-bv* (*bv-or* (*bdd-lookup l w*) (*bdd-lookup r w*)) **by** (*simp add*: *bdd-lookup-binop*)
also from *N* **have** ... = *set-of-bv* (*bdd-lookup l w*) ∪ *set-of-bv* (*bdd-lookup r w*) **by** (*simp add*: *bv-or-set-of-bv*)
also have ... = *set-of-bv* (*bdd-lookup* (*Branch l r*) (*insertl* 0 *False w*)) ∪ *set-of-bv* (*bdd-lookup* (*Branch l r*) (*insertl* 0 *True w*)) **by** (*cases w*) *simp*+
also have ... = (⋃ *b* ∈ {*True*, *False*}. *set-of-bv* (*bdd-lookup* (*Branch l r*) (*insertl* 0 *b w*))) **by** *auto*
also have ... = (⋃ *b*. *set-of-bv* (*bdd-lookup* (*Branch l r*) (*insertl* 0 *b w*))) **by** *blast*

finally show *?case* **.**
next
case (3 *n l r k w*)
then obtain *j* **where** *J*: *k* = *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*)
= (⋃ *i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) *ws*))
(**is** *set-of-bv ?q* = -)
proof (*simp only*: *set-eq-iff*, *rule allI*)
fix *x* :: *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* = *length* (*fst N*) **by** (*simp add*: *nfa-is-node-def*)
{
fix *i* **assume** *H*: *i* < *length* (*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*: ⋀*i*. *i* < *length q* ⟹ *nfa-is-node N* (*bdd-lookup* (*fst N* ! *i*) *ws*) **by** (*simp add*: *nfa-is-node-def*)

from *L assms* **have** *x* ∈ *set-of-bv ?q* = (*x* < *length* (*fst N*) ∧ (∃ *i* ∈ *set-of-bv q*. *bdd-lookup* (*fst N* ! *i*) *ws* ! *x* ∧ *i* < *length q*)) **by** (*auto simp add*: *set-of-bv-def bdd-lookup-subsetbdd*)
also from *I* **have** ... = (*x* ∈ (⋃ *i* ∈ *set-of-bv q*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) *ws*))) **by** (*auto simp*: *nfa-is-node-def set-of-bv-def*)
finally show *x* ∈ *set-of-bv ?q* = (*x* ∈ (⋃ *i* ∈ *set-of-bv q*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) *ws*))) **.**
qed

lemma *nfa-trans-quantify-nfa*:
assumes *wf-nfa N* (*Suc n*)
and *v* ≤ *n*
and *is-alph n w*
and *nfa-is-node N q*
shows *set-of-bv* (*nfa-trans* (*quantify-nfa v N*) *q w*) = (⋃ *b*. *set-of-bv* (*nfa-trans N q* (*insertl v b w*)))
proof −
from *assms* **have** *V*1: *wf-nfa* (*quantify-nfa v N*) *n* **by** (*simp add*: *quantify-nfa-well-formed-aut*)
with *assms* **have** *V*2: *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* ∈ *set-of-bv q*
with *assms* **have** *i* < *length* (*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*: ⋀*i*. *i* ∈ *set-of-bv q* ⟹ *length w* = *n* ∧ *bddh* (*Suc n*) (*fst N* ! *i*) ∧ *bdd-all* (*nfa-is-node N*) (*fst N* ! *i*) ∧ *v* ≤ *n* **by** (*simp add*: *is-alph-def*)
from *assms* **have** *V*3: ⋀*b*. *wf-nfa N* (*length* (*insertl v b w*)) **by** (*simp add*: *wf-nfa-def is-alph-def insertl-len*)
from *N V*2 **have** *set-of-bv* (*bdd-lookup* (*subsetbdd* (*fst* (*quantify-nfa v N*)) *q* (*nfa-emptybdd* (*length q*))) *w*) = (⋃*i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*fst* (*quantify-nfa v N*) ! *i*) *w*))
by (*simp add*: *subsetbdd-set-of-bv*)
also from *assms* **have** ... = (⋃*i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*quantify-bdd v* (*fst N* ! *i*)) *w*)) **by** (*auto simp*: *quantify-nfa-def split-beta nfa-is-node-def set-of-bv-def*)
also have ... = (⋃*i*∈*set-of-bv q*. ⋃*b*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*)))
proof (*simp only*: *set-eq-iff*, *rule allI*)
fix *x*
have *x* ∈ (⋃*i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*quantify-bdd v* (*fst N* ! *i*)) *w*)) = (∃*i*∈*set-of-bv q*. *x* ∈ *set-of-bv* (*bdd-lookup* (*quantify-bdd v* (*fst N* ! *i*)) *w*)) **by** *simp*
also have ... = ({*i*. *i* ∈ *set-of-bv q* ∧ *x* ∈ *set-of-bv* (*bdd-lookup* (*quantify-bdd v* (*fst N* ! *i*)) *w*)} ≠ {}) **by** *auto*
also from *I* **have** ... = ({*i*. *i* ∈ *set-of-bv q* ∧ *x* ∈ (⋃*b*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*)))} ≠ {}) **by** (*auto simp*: *bdd-lookup-quantify-bdd-set-of-bv*[*of w n - N*])
also have ... = (∃*i*∈*set-of-bv q*. *x* ∈ (⋃*b*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*)))) **by** *auto*
also have ... = (*x* ∈ (⋃*i*∈*set-of-bv q*. ⋃*b*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*)))) **by** *simp*
finally show (*x* ∈ (⋃*i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*quantify-bdd v* (*fst N* ! *i*)) *w*))) = (*x* ∈ (⋃*i* ∈ *set-of-bv q*. ⋃*b*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*)))) .
qed
also have ... = (⋃*b*. ⋃*i*∈*set-of-bv q*. *set-of-bv* (*bdd-lookup* (*fst N* ! *i*) (*insertl v b w*))) **by** *auto*
also from *V*3 *assms* **have** ... = (⋃*b*. *set-of-bv* (*bdd-lookup* (*subsetbdd* (*fst N*) *q* (*nfa-emptybdd* (*length q*))) (*insertl v b w*))) **by** (*simp add*: *subsetbdd-set-of-bv*)
finally show *?thesis* **by** (*simp add*: *nfa-trans-def*)
qed

fun *insertll* :: *nat* ⇒ *'a list* ⇒ *'a list list* ⇒ *'a list list*
where
insertll i [] [] = []
| *insertll i* (*a* # *as*) (*bs* # *bss*) = *insertl i a bs* # *insertll i as bss*

lemma *insertll-len2*:
assumes *list-all* (*is-alph n*) *vs*
and *length x* = *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*: *insertl-len is-alph-def*)+

lemma *insertll-append*:
assumes *length xs* = *length vs*
shows *insertll k* (*xs* @ [*x*]) (*vs* @ [*v*]) = *insertll k xs vs* @ [*insertl k x v*]
using *assms* **by** (*induct k xs vs rule*: *insertll.induct*) *simp*+

lemma *UN-UN-lenset*: (⋃ *b*. ⋃ *x*∈{*x*. *length x* = *n*}. *M b x*) = (⋃ *bs*∈{*x*. *length x* = *Suc n*}. *M* (*last bs*) (*butlast bs*))
proof *auto*
fix *x b xa* **assume** *x* ∈ *M b xa*
hence *length* (*xa* @ [*b*]) = *Suc* (*length xa*) ∧ *x* ∈ *M* (*last* (*xa* @ [*b*])) (*butlast* (*xa* @ [*b*])) **by** *simp*
thus ∃ *bs*. *length bs* = *Suc* (*length xa*) ∧ *x* ∈ *M* (*last bs*) (*butlast bs*) **..**
next
fix *x bs* **assume** *x* ∈ *M* (*last bs*) (*butlast bs*) **and** *length bs* = *Suc n*
hence *length* (*butlast bs*) = *n* ∧ *x* ∈ *M* (*last bs*) (*butlast bs*) **by** *simp*
thus ∃ *b xa*. *length xa* = *n* ∧ *x* ∈ *M b xa* **by** *blast*
qed

lemma *nfa-steps-quantify-nfa*:
assumes *wf-nfa N* (*Suc n*)
and *list-all* (*is-alph n*) *w*
and *nfa-is-node N q*
and *v* ≤ *n*
shows *set-of-bv* (*nfa-steps* (*quantify-nfa v N*) *q w*) = (⋃ *xs* ∈ {*x*. *length x* = *length w*}. *set-of-bv* (*nfa-steps N q* (*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-is-node N* (*nfa-steps* (*quantify-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** ⋀ *b*. *length* (*insertl v b x*) = *Suc n* **by** (*simp add*: *insertl-len is-alph-def*)
with *snoc* **have** *B*: ⋀ *b*. *wf-nfa N* (*length* (*insertl v b x*)) **by** *simp*
from *snoc* **have** *IV*: *set-of-bv* (*nfa-steps* (*quantify-nfa v N*) *q xs*) = (⋃ *x* ∈{*x*. *length x* = *length xs*}. *set-of-bv* (*nfa-steps N q* (*insertll v x xs*))) **by** *simp*

{ fix *bs* :: *bool list* **assume** *H*: *length bs* = *length xs*
with *snoc* **have** *list-all* (*is-alph* (*Suc n*)) (*insertll v bs xs*) **by** (*simp add*: *insertll-len*2)
with *snoc* **have** *nfa-is-node N* (*nfa-steps N q* (*insertll v bs xs*)) **by** (*simp add*: *nfa-steps-is-node*)
} note *N*2 = *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 ... = *set-of-bv* (*nfa-trans* (*quantify-nfa v N*) *?q x*) **by** *simp*
also from *snoc N* **have** ... = (⋃ *b*. *set-of-bv* (*nfa-trans N ?q* (*insertl v b x*))) **by** (*simp add*: *nfa-trans-quantify-nfa*)
also have ... = (⋃ *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** ... = (⋃ *b*. ⋃ *i*∈*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** ... = (⋃ *b*. ⋃ *i*∈(⋃ *x*∈{*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 ... = (⋃ *b*. ⋃ *y*∈{*x*. *length x* = *length xs*}. ⋃ *i*∈*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 ... = (⋃ *bs*∈{*x*. *length x* = *Suc* (*length xs*)}. ⋃ *i*∈*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 *N*2 *B* **have** ... = (⋃ *bs*∈{*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 ... = (⋃ *bs*∈{*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 ... = (⋃ *bs*∈{*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 ... = (⋃ *bs*∈{*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* ∈ (⋃ *bs*∈{*x*. *length x* = *Suc* (*length xs*)}. *set-of-bv* (*nfa-steps N q* (*insertll v* (*butlast bs* @ [*last bs*]) (*xs* @ [*x*]))))) =
(∃ *bs* ∈ {*x*. *length x* = *Suc* (*length xs*)}. *bs* ≠ [] ∧ *xa* ∈ *set-of-bv* (*nfa-steps N q* (*insertll v* (*butlast bs* @ [*last bs*]) (*xs* @ [*x*])))) **by** *auto*
also have ... = (∃ *bs* ∈ {*x*. *length x* = *Suc* (*length xs*)}. *bs* ≠ [] ∧ *xa* ∈ *set-of-bv* (*nfa-steps N q* (*insertll v bs* (*xs* @ [*x*])))) **by** *auto*
also have ... = (*xa* ∈ (⋃ *bs*∈{*x*. *length x* = *Suc* (*length xs*)}. *set-of-bv* (*nfa-steps N q* (*insertll v bs* (*xs* @ [*x*]))))) **by** *auto*
finally show (*xa* ∈ (⋃ *bs*∈{*x*. *length x* = *Suc* (*length xs*)}. *set-of-bv* (*nfa-steps*

N q (*insertll v* (*butlast bs* @ [*last bs*]) (*xs* @ [*x*])))) =
(*xa* ∈ (⋃ *bs*∈{*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* ≤ *n*
and *list-all* (*is-alph n*) *bss*
shows *nfa-accepts* (*quantify-nfa i A*) *bss* = (∃ *bs*. *nfa-accepts A* (*insertll i bs bss*) ∧ *length bs* = *length bss*)
proof −
note *Q*0 = *nfa-startnode-is-node*[*OF assms*(1)]
hence *nfa-is-node A* (*nfa-startnode* (*quantify-nfa i A*)) **by** (*simp add*: *nfa-startnode-def quantify-nfa-def split-beta*)
with *assms* **have** *I*: *set-of-bv* (*nfa-steps* (*quantify-nfa i A*) (*nfa-startnode* (*quantify-nfa i A*)) *bss*) =
(⋃ *bs* ∈ {*bs*. *length bs* = *length bss*}. *set-of-bv* (*nfa-steps A* (*nfa-startnode* (*quantify-nfa i A*)) (*insertll i bs bss*)))
by (*simp add*: *nfa-steps-quantify-nfa*)
have *nfa-accepts* (*quantify-nfa i A*) *bss* = *nfa-accepting* (*quantify-nfa i A*) (*nfa-steps* (*quantify-nfa i A*) (*nfa-startnode* (*quantify-nfa i A*)) *bss*) **by** (*simp add*: *accepts-def*)
also have ... = (*set-of-bv* (*snd* (*quantify-nfa i A*)) ∩ *set-of-bv* (*nfa-steps* (*quantify-nfa i A*) (*nfa-startnode* (*quantify-nfa i A*)) *bss*) ≠ {}) **by** (*simp add*: *nfa-accepting-set-of-bv*)
also from *I* **have** ... = (*set-of-bv* (*snd A*) ∩ (⋃ *bs* ∈ {*bs*. *length bs* = *length bss*}. *set-of-bv* (*nfa-steps A* (*nfa-startnode* (*quantify-nfa i A*)) (*insertll i bs bss*))) ≠ {})
by (*simp add*: *quantify-nfa-def split-beta*)
also have ... = ((⋃ *bs* ∈ {*bs*. *length bs* = *length bss*}. *set-of-bv* (*snd A*) ∩ *set-of-bv* (*nfa-steps A* (*nfa-startnode* (*quantify-nfa i A*)) (*insertll i bs bss*))) ≠ {}) **by** *simp*
also have ... = (∃ *bs* ∈ {*bs*. *length bs* = *length bss*}. *set-of-bv* (*snd A*) ∩ *set-of-bv* (*nfa-steps A* (*nfa-startnode A*) (*insertll i bs bss*)) ≠ {}) **by** (*auto simp*: *nfa-startnode-def quantify-nfa-def split-beta*)
also have ... = (∃ *bs*. *nfa-accepts A* (*insertll i bs bss*) ∧ *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* × *bool list* ⇒ *nat* ⇒ *nat* ⇒ *nat list* **where**
rquot-succs M = (λ*n x*. [*bdd-lookup* (*fst M* ! *x*) (*replicate n False*)])

definition
rquot-invariant :: *nat bdd list* × *bool list* ⇒ *bool list* ⇒ *bool* **where**

rquot-invariant M = (λ*l*. *length l* = *length* (*fst M*))

definition
rquot-ins = (λ*x l*. *l*[*x*:=*True*])

definition
rquot-memb :: *nat* ⇒ *bool list* ⇒ *bool* **where**
rquot-memb = (λ*x l*. *l* ! *x*)

definition
rquot-empt :: *nat bdd list* × *bool list* ⇒ *bool list* **where**
rquot-empt M = *replicate* (*length* (*fst M*)) *False*

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

definition
zeros :: *nat* ⇒ *nat* ⇒ *bool list list* **where**
zeros m n = *replicate m* (*replicate n False*)

lemma *zeros-is-alpha*: *list-all* (*is-alph v*) (*zeros n v*)
by (*induct n*) (*simp add*: *zeros-def is-alph-def*)+

lemma *zeros-rone*: *zeros* (*Suc n*) *v* = *zeros n v* @ *zeros* 1 *v*
by (*simp add*: *zeros-def replicate-append-same*)

lemma *zeros-len*: *length* (*zeros n v*) = *n*
by (*simp add*: *zeros-def*)

lemma *zeros-rtrancl*: (∃ *n*. *dfa-reach M x* (*zeros n v*) *y*) = ((*x*,*y*) ∈ (*succsr* (*rquot-succs M v*))*)
proof
assume ∃ *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*: ∃ *n*. *w* = *zeros n v* **by** *auto*
from *w-def N* **have** *dfa-reach M x w y* **by** *simp*
from *this W* **show** (*x*,*y*) ∈ (*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 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 only*: *zeros-rone*)
also have ... = *zeros n v* @ [*replicate v False*] **by** (*simp add*: *zeros-def*)
finally have *zeros n*′ *v* = *zeros n v* @ [*replicate v False*] **.**
with *N*′ **have** *WS*: *ws* = *zeros n v y* = *replicate v False* **by** *auto*

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

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

lemma *map-index-len*:
$length\ (map\text{-}index\ f\ ls\ n) = length\ ls$
by (*induct ls arbitrary*: *n*) *simp+*

lemma *map-index-nth*:
assumes $i < length\ l$
shows $map\text{-}index\ f\ l\ n\ !\ i = f\ (l\ !\ i)\ (n + i)$
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*: $i = Suc\ j$ **by** (*cases i*) *simp+*
with *Cons* **show** *?thesis* **by** *simp*
qed *simp*
qed *simp*

definition

rquot :: *dfa* ⇒ *nat* ⇒ *dfa* **where**
rquot = (λ(*bd*, *as*) *v*. (*bd*, *map-index* (λ*x n*. *nfa-accepting*′ *as* (*rquot-dfs* (*bd*, *as*) *v n*)) *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** ¬ *rquot-memb y S*
thus *rquot-memb x* (*rquot-ins y S*) = (*x* = *y* ∨ *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*) = (∃ *m*. *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 ... = (∃ *m*. *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* = (∃ *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*) ∩ *set-of-bv* (*rquot-dfs A n q*) ≠ {}) **by** (*simp add*: *nfa-accepting′-set-of-bv*)
 also have ... = (∃ *i*. *i* < *length* (*snd A*) ∧ *snd A* ! *i* ∧ *i* < *length* (*rquot-dfs A n q*) ∧ *rquot-dfs A n q* ! *i*) **by** (*auto simp*: *set-of-bv-def*)
 also from *well-formed L* **have** ... = (∃ *i*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ *rquot-memb i* (*rquot-dfs A n q*)) **by** (*auto simp add*: *wf-dfa-def dfa-is-node-def rquot-memb-def*)
 also have ... = ({*i*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ *rquot-memb i* (*rquot-dfs A n q*)} ≠ {}) **by** *auto*
 also from *assms Q* **have** ... = ({*i*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ (∃ *m*. *dfa-reach A q* (*zeros m n*) *i*)} ≠ {}) **by** (*auto simp*: *dfa-reach-rquot*)
 also have ... = ({*i*. ∃ *m*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ *i* = *dfa-steps A q* (*zeros m n*)} ≠ {}) **by** (*simp add*: *reach-def*)
 also have ... = (∃ *i m*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ *i* = *dfa-steps A q* (*zeros m n*)) **by** *auto*
 also have ... = (∃ *m*. *snd A* ! *dfa-steps A q* (*zeros m n*))
 proof
  assume ∃ *m*. *snd A* ! *dfa-steps A q* (*zeros m 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** ∃ *i m*. *dfa-is-node A i* ∧ *snd A* ! *i* ∧ *i* = *dfa-steps A q* (*zeros m n*) **by** *auto*
 qed *auto*
 finally have *nfa-accepting′* (*snd A*) (*rquot-dfs A n q*) = (∃ *m*. *snd A* ! *dfa-steps A q* (*zeros m 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* = (∃ *m*. *dfa-accepts A* (*bss* @ *zeros m 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 *dfa-accepts* (*rquot A n*) *bss* = *dfa-accepting* (*rquot A n*) (*dfa-steps* (*rquot A n*) 0 *bss*) **by** (*simp add*: *accepts-def*)
also from *assms q* **have** ... = (∃ *m*. *dfa-accepting A* (*dfa-steps A* (*dfa-steps A* 0 *bss*) (*zeros m n*))) **by** (*simp add*: *rquot-accepting rquot-steps*)
also have ... = (∃ *m*. *dfa-accepting A* (*dfa-steps A* 0 (*bss* @ *zeros m n*))) **by** *simp*
also have ... = (∃ *m*. *dfa-accepts A* (*bss* @ *zeros m n*)) **by** (*simp add*: *accepts-def*)
finally show *?thesis* **.**
qed

5.7 Diophantine Equations

fun *eval-dioph* :: *int list* ⇒ *nat list* ⇒ *int*
where
eval-dioph (*k* # *ks*) (*x* # *xs*) = *k* ∗ *int x* + *eval-dioph ks xs*
| *eval-dioph ks xs* = 0

lemma *eval-dioph-mult*:
eval-dioph ks xs ∗ *int n* = *eval-dioph ks* (*map* (λ*x*. *x* ∗ *n*) *xs*)
by(*induct ks xs rule*: *eval-dioph.induct*) (*simp-all add*: *distrib-right*)

lemma *eval-dioph-add-map*:
eval-dioph ks (*map f xs*) + *eval-dioph ks* (*map g xs*) =
eval-dioph ks (*map* (λ*x*. *f x* + *g x*) (*xs*::*nat list*))
proof (*induct ks xs rule*: *eval-dioph.induct*)
case (1 *k ks x xs*)
have *eval-dioph* (*k* # *ks*) (*map f* (*x* # *xs*)) + *eval-dioph* (*k* # *ks*) (*map g* (*x* # *xs*)) =
(*k* ∗ *int* (*f x*) + *k* ∗ *int* (*g x*)) + (*eval-dioph ks* (*map f xs*) + *eval-dioph ks* (*map g xs*))
by *simp*
also have ... = (*k* ∗ *int* (*f x*) + *k* ∗ *int* (*g x*)) + *eval-dioph ks* (*map* (λ*x*. *f x* + *g x*) *xs*)
by (*simp add*: 1)
finally show *?case* **by** (*simp add*: *ac-simps distrib-left*)
qed *simp-all*

lemma *eval-dioph-div-mult*:
eval-dioph ks (*map* (λ*x*. *x div n*) *xs*) ∗ *int n* +
eval-dioph ks (*map* (λ*x*. *x mod n*) *xs*) = *eval-dioph ks xs*

by (*simp add*: *eval-dioph-mult o-def eval-dioph-add-map*)

lemma *eval-dioph-mod*:
eval-dioph ks xs mod int n = eval-dioph ks (*map* (λ*x*. *x mod n*) *xs*) *mod int n*
proof (*induct ks xs rule*: *eval-dioph.induct*)
case (1 *k ks x xs*)
have *eval-dioph* (*k # ks*) (*x # xs*) *mod int n* =
((*k* ∗ *int x*) *mod int n* + *eval-dioph ks xs mod int n*) *mod int n*
by (*simp add*: *mod-add-eq*)
also have ... = ((*k* ∗ (*int x mod int n*)) *mod int n* +
eval-dioph ks (*map* (λ*x*. *x mod n*) *xs*) *mod int n*) *mod 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*:
(*eval-dioph ks xs* = *l*) =
(*eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*) *mod* 2 = *l mod* 2 ∧
eval-dioph ks (*map* (λ*x*. *x div* 2) *xs*) =
(*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *div* 2) (**is** *?l* = *?r*)
proof
assume *eq*: *?l*
then have *eval-dioph ks xs mod* 2 = *l mod* 2 **by** *simp*
with *eval-dioph-mod* [*of* - - 2]
have *eq′*: *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*) *mod* 2 = *l mod* 2
by *simp*
from *eval-dioph-div-mult* [*symmetric*, *of ks xs* 2] *eq*
have *eval-dioph ks* (*map* (λ*x*. *x div* 2) *xs*) ∗ 2 + *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*) = *l*
by *simp*
then have *eval-dioph ks* (*map* (λ*x*. *x div* 2) *xs*) ∗ 2 = *l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)
by (*simp add*: *eq-diff-eq*)
then have (*eval-dioph ks* (*map* (λ*x*. *x div* 2) *xs*) ∗ 2) *div* 2 =
(*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *div* 2
by *simp*
with *eq′* **show** *?r* **by** *simp*
next
assume *?r* (**is** *?r*1 ∧ *?r*2)
then obtain *eq*1: *?r*1 **and** *eq*2: *?r*2 **..**
from *eq*1 **have** (*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*) *mod* 2) *mod* 2 =
(*l* − *l mod* 2) *mod* 2
by *simp*
then have (*l mod* 2 − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*) *mod* 2 *mod* 2) *mod* 2 =
(*l mod* 2 − *l mod* 2 *mod* 2) *mod* 2
by (*simp only*: *mod-diff-eq*)
then have *eq*1′: (*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *mod* 2 = 0
by (*simp add*: *mod-diff-eq*)

```
  from eq2 have
    eval-dioph ks (map (λx. x div 2) xs) * 2 +
     (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2 =
     (l − eval-dioph ks (map (λx. x mod 2) xs)) div 2 * 2 +
     (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2
    by simp
  then have
    eval-dioph ks (map (λx. x div 2) xs) * 2 +
     (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2 =
     l − eval-dioph ks (map (λx. 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 ≤ l) =
   (eval-dioph ks (map (λx. x div 2) xs) ≤
      (l − eval-dioph ks (map (λx. x mod 2) xs)) div 2) (is ?l = ?r)
proof
  assume ?l
  with eval-dioph-div-mult [symmetric, of ks xs 2]
  have eval-dioph ks (map (λx. x div 2) xs) * 2 + eval-dioph ks (map (λx. x mod
2) xs) ≤ l
    by simp
  then have eval-dioph ks (map (λx. x div 2) xs) * 2 ≤ l − eval-dioph ks (map
(λx. x mod 2) xs)
    by (simp add: le-diff-eq)
  then have (eval-dioph ks (map (λx. x div 2) xs) * 2) div 2 ≤
    (l − eval-dioph ks (map (λx. x mod 2) xs)) div 2
    by (rule zdiv-mono1) simp
  then show ?r by simp
next
  assume ?r
  have eval-dioph ks xs ≤ eval-dioph ks xs +
    (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2
    by simp
  also {
  from ‹?r› have eval-dioph ks (map (λx. x div 2) xs) * 2 ≤
    (l − eval-dioph ks (map (λx. x mod 2) xs)) div 2 * 2
    by simp
  also have ... = l − eval-dioph ks (map (λx. x mod 2) xs) −
    (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2
    by (simp add: eq-diff-eq)
  finally have (eval-dioph ks (map (λx. x div 2) xs) * 2 +
    eval-dioph ks (map (λx. x mod 2) xs)) +
    (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2 ≤ l
    by simp
  with eval-dioph-div-mult [of - 2]
```

have *eval-dioph ks xs* +
(*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *mod* 2 ≤ *l*
by *simp* **}**
finally show *?l* **.**
qed

lemma *sum-list-abs-ge*-0: (0::*int*) ≤ *sum-list* (*map abs ks*)
by (*induct ks*) *simp-all*

lemma *zmult-div-aux*1:
assumes *b*: *b* ≠ 0
shows (*a* − *a mod b*) *div b* = (*a*::*int*) *div b*
proof −
from *minus-mod-eq-mult-div* [*symmetric*, *of b a*]
have (*b* ∗ (*a div b*)) *div b* = (*a* − *a mod b*) *div b*
by *simp*
with *b* **show** *?thesis* **by** *simp*
qed

lemma *zmult-div-aux*2:
assumes *b*: *b* ≠ 0
shows ((*a*::*int*) − *a mod b*) *mod b* = 0
using *b minus-mod-eq-mult-div* [*symmetric*, *of b a*, *symmetric*]
by *simp*

lemma *div-abs-eq*:
assumes *mod*: (*a*::*int*) *mod b* = 0
and *b*: 0 < *b*
shows |*a div b*| = |*a*| *div b*
proof (*cases* 0 ≤ *a*)
case *True* **with** *pos-imp-zdiv-nonneg-iff* [*OF b*]
show *?thesis* **by** *auto*
next
from *b* **have** *b* ≠ 0 **by** *auto*
case *False*
then have *a* < 0 **by** *auto*
have |*a div b*| = − (*a div b*)
by (*simp add*: *div-neg-pos-less*0 [*OF* ‹*a* < 0› *b*] *zabs-def*)
with *abs-of-neg* [*OF* ‹*a* < 0›] *zdiv-zminus*1-*eq-if* [*OF* ‹*b* ≠ 0›] *mod*
show *?thesis* **by** *simp*
qed

lemma *add-div-trivial*: 0 ≤ *c* ⟹ *c* < *b* ⟹ ((*a*::*int*) ∗ *b* + *c*) *div b* = *a*
by (*simp add*: *div-add*1-*eq div-pos-pos-trivial*)

lemma *dioph-rhs-bound*:
|(*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *div* 2| ≤ *max* |*l*| (∑ *k*←*ks*. |*k*|)
proof −
have |(*l* − *eval-dioph ks* (*map* (λ*x*. *x mod* 2) *xs*)) *div* 2| =

```
  |(l − eval-dioph ks (map (λx. x mod 2) xs) −
   (l − eval-dioph ks (map (λx. x mod 2) xs)) mod 2) div 2|
  (is - = |(- − ?r) div 2|)
  by (simp add: zmult-div-aux1)
also have ... = |l − eval-dioph ks (map (λx. x mod 2) xs) − ?r| div 2
  by (simp add: zmult-div-aux2 div-abs-eq)
also have |l − eval-dioph ks (map (λx. x mod 2) xs) − ?r| ≤
  |l − eval-dioph ks (map (λx. x mod 2) xs)| + |?r|
  by (rule abs-triangle-ineq4)
also have |l − eval-dioph ks (map (λx. x mod 2) xs)| ≤
  |l| + |eval-dioph ks (map (λx. x mod 2) xs)|
  by (rule abs-triangle-ineq4)
also have |eval-dioph ks (map (λx. x mod 2) xs)| ≤ (∑k←ks. |k|)
proof (induct ks xs rule: eval-dioph.induct)
  case (1 k ks x xs)
  have |k * int (x mod 2) + eval-dioph ks (map (λx. x mod 2) xs)| ≤
    |k * int (x mod 2)| + |eval-dioph ks (map (λx. x mod 2) xs)|
    by (rule abs-triangle-ineq)
  also have |k * int (x mod 2)| ≤ |k| * |int (x mod 2)|
    by (simp add: abs-mult)
  also have |int (x mod 2)| ≤ 1 by simp
  finally have |k * int (x mod 2) + eval-dioph ks (map (λx. x mod 2) xs)| ≤
    |k| + |eval-dioph ks (map (λx. x 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 − eval-dioph ks (map (λx. x mod 2) xs)) div 2| ≤
  (|l| + (∑k←ks. |k|) + |?r|) div 2
  by (simp add: zdiv-mono1)
show ?thesis
proof (cases (∑k←ks. |k|) ≤ |l|)
  case True
  note ineq
  also from True
  have (|l| + (∑k←ks. |k|) + |?r|) div 2 ≤ (|l| * 2 + |?r|) div 2
    by (simp add: zdiv-mono1)
  also have ... = |l|
    by (simp add: add-div-trivial)
  finally show ?thesis by simp
next
  case False
  note ineq
  also from False
  have (|l| + (∑k←ks. |k|) + |?r|) div 2 ≤ ((∑k←ks. |k|) * 2 + |?r|) div 2
    by (simp add: zdiv-mono1)
  also have ... = (∑k←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 $|(m - eval\text{-}dioph\ ks\ (map\ (\lambda x.\ x\ mod\ 2)\ xs))\ div\ 2| \leq max\ |l|\ (\sum k{\leftarrow}ks.\ |k|)$
proof (*cases* $(\sum k{\leftarrow}ks.\ |k|) \leq |l|$)
case *True*
have $|(m - eval\text{-}dioph\ ks\ (map\ (\lambda x.\ x\ mod\ 2)\ xs))\ div\ 2| \leq max\ |m|\ (\sum k{\leftarrow}ks.\ |k|)$
by (*rule dioph-rhs-bound*)
also from *True m* **have** $|m| \leq |l|$ **by** *simp*
finally show *?thesis* **by** *simp*
next
case *False*
have $|(m - eval\text{-}dioph\ ks\ (map\ (\lambda x.\ x\ mod\ 2)\ xs))\ div\ 2| \leq max\ |m|\ (\sum k{\leftarrow}ks.\ |k|)$
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*: $\forall (i::int) \in S.\ |i| < j$
shows *finite S*
proof (*rule finite-subset*)
have *finite* $(int\ `\ \{n.\ n < nat\ j\})$
by (*rule nat-seg-image-imp-finite* [*OF refl*])
moreover have *finite* $((\lambda n.\ -\ int\ n)\ `\ \{n.\ n < nat\ j\})$
by (*rule nat-seg-image-imp-finite* [*OF refl*])
ultimately show *finite* $(int\ `\ \{n.\ n < nat\ j\} \cup (\lambda n.\ -\ int\ n)\ `\ \{n.\ n < nat\ j\})$
by (*rule finite-UnI*)
show $S \subseteq int\ `\ \{n.\ n < nat\ j\} \cup (\lambda n.\ -\ int\ n)\ `\ \{n.\ n < nat\ j\}$
proof
fix *i*
assume *i*: $i \in S$
show $i \in int\ `\ \{n.\ n < nat\ j\} \cup (\lambda n.\ -\ int\ n)\ `\ \{n.\ n < nat\ j\}$
proof (*cases* $0 \leq i$)
case *True*
then have $i = int\ (nat\ i)$ **by** *simp*
moreover from *i S* **have** $nat\ i \in \{n.\ n < nat\ j\}$
by *auto*
ultimately have $i \in int\ `\ \{n.\ n < nat\ j\}$
by (*rule image-eqI*)
then show *?thesis* **..**
next
case *False*

```
      then have i = - int (nat (- i)) by simp
      moreover from i S have nat (- i) ∈ {n. n < nat j}
        by auto
      ultimately have i ∈ (λn. - int n) ` {n. n < nat j}
        by (rule image-eqI)
      then show ?thesis ..
    qed
  qed
qed

primrec mk-nat-vecs :: nat ⇒ 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: ∀ xs∈set (mk-nat-vecs n). ∀ x∈set xs. x < 2
  by (induct n) (auto simp add: Let-def)

lemma mk-nat-vecs-mod-eq: xs ∈ set (mk-nat-vecs n) ⟹ map (λx. x mod 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 (λxs.
    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 ⇒ int ⇒ int ⇒ bool where
  dioph-is-node ks l m = (|m| ≤ max |l| (∑ k←ks. |k|))

definition
  dioph-invariant :: int list ⇒ int ⇒ nat option list × int list ⇒ bool where
  dioph-invariant ks l = (λ(is, js). length is = nat (2 * max |l| (∑ k←ks. |k|) +
1))

definition
  dioph-ins m = (λ(is, js). (is[int-encode m := Some (length js)], js @ [m]))

definition
  dioph-memb :: int ⇒ nat option list × int list ⇒ bool where
  dioph-memb m = (λ(is, js). is ! int-encode m ≠ None)

definition
```

```
  dioph-empt :: int list ⇒ int ⇒ nat option list × int list where
  dioph-empt ks l = (replicate (nat (2 * max |l| (∑ k←ks. |k|) + 1)) None, [])

lemma int-encode-bound: dioph-is-node ks l m ⟹
  int-encode m < nat (2 * max |l| (∑ k←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* (λ*j*. *make-bdd* (λ*xs*.
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 (λ*j*. *j* = 0) *js* @ [*False*]))

abbreviation (*input*) *nat-of-bool* :: *bool* ⇒ *nat*
where
nat-of-bool ≡ *of-bool*

lemma *nat-of-bool-bound*: *nat-of-bool b* < 2
by (*cases b*) *simp-all*

lemma *nat-of-bool-mk-nat-vecs*:
length bs = *n* ⟹ *map nat-of-bool bs* ∈ *set* (*mk-nat-vecs n*)
apply (*induct n arbitrary*: *bs*)
apply *simp*
apply (*case-tac bs*)
apply *simp*
apply (*case-tac a*)
apply (*simp-all add*: *Let-def*)
done

lemma *bdd-lookup-make-bdd*:
length bs = *n* ⟹ *bdd-lookup* (*make-bdd f n xs*) *bs* = *f* (*xs* @ *map nat-of-bool bs*)
apply (*induct n arbitrary*: *bs xs*)
apply *simp*
apply (*case-tac bs*)
apply *auto*
done

primrec *nat-of-bools* :: *bool list* ⇒ *nat*
where
nat-of-bools [] = 0
| *nat-of-bools* (*b* # *bs*) = *nat-of-bool b* + 2 ∗ *nat-of-bools bs*

primrec *nats-of-boolss* :: *nat* ⇒ *bool list list* ⇒ *nat list*

where
  Nil: *nats-of-boolss n* [] = *replicate n* 0
| *Cons*: *nats-of-boolss n* (*bs* # *bss*) =
    map (λ(*b*, *x*). *nat-of-bool b* + 2 ∗ *x*) (*zip bs* (*nats-of-boolss n bss*))

lemma *nats-of-boolss-length*:
  list-all (*is-alph n*) *bss* ⟹ *length* (*nats-of-boolss n bss*) = *n*
  by (*induct bss*) (*simp-all add*: *is-alph-def*)

lemma *nats-of-boolss-mod*2:
  assumes *bs*: *length bs* = *n* **and** *bss*: *list-all* (*is-alph n*) *bss*
  shows *map* (λ*x*. *x mod* 2) (*nats-of-boolss n* (*bs* # *bss*)) = *map nat-of-bool bs*
proof −
  from *bs bss*
  have *map nat-of-bool* (*map fst* (*zip bs* (*nats-of-boolss n bss*))) = *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-div*2:
  assumes *bs*: *length bs* = *n* **and** *bss*: *list-all* (*is-alph n*) *bss*
  shows *map* (λ*x*. *x div* 2) (*nats-of-boolss n* (*bs* # *bss*)) = *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*: *length xs* = *length ys* ⟹
  zip (*insertl n x xs*) (*insertl n y ys*) = *insertl n* (*x*, *y*) (*zip xs ys*)
  by (*induct n x xs arbitrary*: *ys rule*: *insertl.induct*)
    (*auto simp add*: *Suc-length-conv*)

lemma *map-insertl*: *map f* (*insertl i x xs*) = *insertl i* (*f x*) (*map f xs*)
  by (*induct i x xs rule*: *insertl.induct*) *simp-all*

lemma *insertl-replicate*: *m* ≤ *n* ⟹
  insertl m x (*replicate n x*) = *x* # *replicate n x*
  apply (*induct n arbitrary*: *m*)
  apply *simp*
  apply (*case-tac m*)
  apply *simp-all*
  done

lemma *nats-of-boolss-insertll*:
  list-all (*is-alph n*) *bss* ⟹ *length bs* = *length bss* ⟹ *i* ≤ *n* ⟹
  nats-of-boolss (*Suc n*) (*insertll i bs bss*) = *insertl i* (*nat-of-bools bs*) (*nats-of-boolss n bss*)
  by (*induct i bs bss rule*: *insertll.induct*)
    (*simp-all add*: *zip-insertl nats-of-boolss-length insertll-len*2 *is-alph-def*

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

lemma *zip-replicate-map*: *length xs* = *n* ⟹ *zip* (*replicate n x*) *xs* = *map* (*Pair x*) *xs*
apply (*induct n arbitrary*: *xs*)
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

lemma *zip-replicate-mapr*: *length xs* = *n* ⟹ *zip xs* (*replicate n x*) = *map* (λ*y*. (*y*, *x*)) *xs*
apply (*induct n arbitrary*: *xs*)
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

lemma *zip-assoc*: *map f* (*zip xs* (*zip ys zs*)) = *map* (λ((*x*, *y*), *z*). *f* (*x*, (*y*, *z*))) (*zip* (*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*:
list-all (*is-alph n*) *bss* ⟹ *list-all* (*is-alph n*) *bss′* ⟹
nats-of-boolss n (*bss* @ *bss′*) =
map (λ(*x*, *y*). *x* + 2 ^ *length bss* ∗ *y*) (*zip* (*nats-of-boolss n bss*) (*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-map*2 *zip-assoc is-alph-def*)

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

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

fun *bools-of-nat* :: *nat* ⇒ *nat* ⇒ *bool list*
where
bools-of-nat k n =
(*if n* = 0 *then*
(*if k* = 0 *then* [] *else False* # *bools-of-nat* (*k* − 1) *n*)
else (*n mod* 2 = 1) # *bools-of-nat* (*k* − 1) (*n div* 2))

lemma *bools-of-nat-length*: $k \leq$ *length* (*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.empt-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*: *fst z* ! *i* = *Some k*
shows *fst* (*gen-dfs* (*dioph-succs n ks*) *dioph-ins dioph-memb z xs*) ! *i* = *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* = *int-encode x*)
apply *simp-all*
done

lemma *dioph-dfs-start*:
fst (*dioph-dfs n ks l*) ! *int-encode l* = *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*: ¬ *dfa-accepting* (*eq-dfa n ks l*) (*dfa-steps* (*eq-dfa n ks l*) (*length* (*snd* (*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*) ∈ (*succsr* (*dioph-succs n ks*))* ⟹ *list-all* (*is-alph n*) *bss* ⟹
dfa-accepting (*eq-dfa n ks l*) (*dfa-steps* (*eq-dfa n ks l*) (*the* (*fst* (*dioph-dfs n ks l*) ! *int-encode m*)) *bss*) =
(*eval-dioph ks* (*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 ‹(*l*, *m*) ∈ (*succsr* (*dioph-succs n ks*))*›

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*: *fst* (*dioph-dfs n ks l*) ! *int-encode m* = *Some k*
by (*auto simp add*: *dioph-memb-def*)
with *m* **have** *k* < *length* (*snd* (*dioph-dfs n ks l*)) ∧ (*snd* (*dioph-dfs n ks 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*: *dioph-is-node ks l l*
by (*simp add*: *dioph-is-node-def*)
with ‹(*l*, *m*) ∈ (*succsr* (*dioph-succs n ks*))*›
have *m*: *dioph-is-node ks l m*
by (*rule dioph-dfs.succsr-is-node*)
with *l Cons*
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*: *fst* (*dioph-dfs n ks l*) ! *int-encode m* = *Some k*
by (*auto simp add*: *dioph-memb-def*)
with *m* **have** *k'*: *k* < *length* (*snd* (*dioph-dfs n ks l*)) ∧ (*snd* (*dioph-dfs n ks 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' Cons*
have *bdd-lookup* (*fst* (*eq-dfa n ks l*) ! *k*) *bs* =
the (*fst* (*dioph-dfs n ks l*) ! *int-encode* ((*m* − *eval-dioph ks* (*map nat-of-bool bs*)) *div* 2))
by (*simp add*: *eq-dfa-def split-beta nth-append bdd-lookup-make-bdd is-alph-def*)
moreover have (*l*, (*m* − *eval-dioph ks* (*map nat-of-bool bs*)) *div* 2) ∈ (*succsr* (*dioph-succs n 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-div*2 *nats-of-boolss-mod*2 *is-alph-def dfa-trans-def* [*abs-def*])
next

```
    case False
    with k' Cons
    have bdd-lookup (fst (eq-dfa n ks l) ! k) bs = length (snd (dioph-dfs n ks 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-alph 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 ∀ x∈set (snd (dioph-dfs n ks l)). ∀ ys∈set (mk-nat-vecs n).
    eval-dioph ks ys mod 2 = x mod 2 ⟶
      the (fst (dioph-dfs n ks l) ! int-encode ((x − eval-dioph ks ys) div 2)) <
        Suc (length (snd (dioph-dfs n ks l)))
  proof (intro ballI impI)
    fix x ys
    assume x: x ∈ set (snd (dioph-dfs n ks l))
    and ys: ys ∈ set (mk-nat-vecs n)
    and ys': eval-dioph ks ys mod 2 = x mod 2
    from x obtain k where k: fst (dioph-dfs n ks l) ! int-encode x = 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) ∈ (succsr (dioph-succs n ks))* using k'
      by (simp add: dioph-dfs.dfs-eq-rtrancl dioph-dfs-def)
    then have (l, (x − eval-dioph ks ys) div 2) ∈ (succsr (dioph-succs n 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′*: *fst* (*dioph-dfs n 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* (λ*xs*.
(*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*←*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* (λ*j*. *make-bdd* (λ*xs*.
the (*is* ! *int-encode* ((*j* − *eval-dioph ks xs*) *div* 2))) *n* []) *js*,
map (λ*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.empt-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-succs 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.empt 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.empt-invariant)
  apply (simp add: dioph-ineq-dfs.empt dioph-is-node-def)
  apply (simp add: dioph-ineq-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 ineq-dfa-accepting:
  (l, m) ∈ (succsr (dioph-ineq-succs n ks))* ⟹ list-all (is-alph n) bss ⟹
  dfa-accepting (ineq-dfa n ks l) (dfa-steps (ineq-dfa n ks l) (the (fst (dioph-ineq-dfs
n ks l) ! int-encode m)) bss) =
  (eval-dioph ks (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 ‹(l, m) ∈ (succsr (dioph-ineq-succs n ks))*›
  have m: dioph-is-node ks l m
    by (rule dioph-ineq-dfs.succsr-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 < *length* (*snd* (*dioph-ineq-dfs n ks l*)) ∧ (*snd* (*dioph-ineq-dfs n 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 ‹(*l*, *m*) ∈ (*succsr* (*dioph-ineq-succs n ks*))*›
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 < *length* (*snd* (*dioph-ineq-dfs n ks l*)) ∧ (*snd* (*dioph-ineq-dfs n 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* − *eval-dioph ks* (*map nat-of-bool bs*)) *div* 2) ∈ (*succsr* (*dioph-ineq-succs n 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-div*2 *nats-of-boolss-mod*2 *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*) ≤ *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 ∀ *x*∈*set* (*snd* (*dioph-ineq-dfs n ks l*)). ∀ *ys*∈*set* (*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* ∈ *set* (*snd* (*dioph-ineq-dfs n ks l*))
and *ys*: *ys* ∈ *set* (*mk-nat-vecs n*)
from *x* **obtain** *k* **where** *k*: *fst* (*dioph-ineq-dfs n ks l*) ! *int-encode x* = *Some k*
and *k'*: *dioph-is-node ks l x*
by (*auto simp add*: *in-set-conv-nth dioph-ineq-dfs-bij* [*symmetric*])
from *k* **have** *dioph-memb x* (*dioph-ineq-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*) ∈ (*succsr* (*dioph-ineq-succs n ks*))* **using** *k'*
by (*simp add*: *dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def*)
then have (*l*, (*x* − *eval-dioph ks ys*) *div* 2) ∈ (*succsr* (*dioph-ineq-succs n 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 *dioph-ineq-dfs.succs-is-node* [*OF k'*, *of n*] *ys*
have *x'*: *dioph-is-node ks l* ((*x* − *eval-dioph ks ys*) *div* 2)
by (*simp add*: *dioph-ineq-succs-def list-all-iff*)
ultimately have *dioph-memb* ((*x* − *eval-dioph ks ys*) *div* 2) (*dioph-ineq-dfs n ks l*)
by (*simp add*: *dioph-ineq-dfs.dfs-eq-rtrancl dioph-ineq-dfs-def ll*)
then obtain *k'* **where** *k'*: *fst* (*dioph-ineq-dfs n 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-ineq-dfs n ks l*)) ∧
snd (*dioph-ineq-dfs n ks l*) ! *k'* = ((*x* − *eval-dioph ks ys*) *div* 2)
by (*simp add*: *dioph-ineq-dfs-bij* [*symmetric*])
with *k'*
show *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*))
by *simp*
qed
moreover have *fst* (*dioph-ineq-dfs n ks l*) ! *int-encode l* = *Some* 0 ∧ *dioph-is-node ks l l*
by (*simp add*: *dioph-ineq-dfs-start dioph-is-node-def*)
then have *snd* (*dioph-ineq-dfs n ks l*) ≠ []
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 bddh-make-bdd bdd-all-make-bdd*)
qed

6 Presburger Arithmetic

datatype *pf* =
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* ⇒ *passign* ⇒ *bool*
where
eval-pf (*Eq ks l*) *xs* = (*eval-dioph ks xs* = *l*)
| *eval-pf* (*Le ks l*) *xs* = (*eval-dioph ks xs* ≤ *l*)
| *eval-pf* (*And p q*) *xs* = (*eval-pf p xs* ∧ *eval-pf q xs*)
| *eval-pf* (*Or p q*) *xs* = (*eval-pf p xs* ∨ *eval-pf q xs*)
| *eval-pf* (*Imp p q*) *xs* = (*eval-pf p xs* ⟶ *eval-pf q xs*)
| *eval-pf* (*Forall p*) *xs* = (∀ *x*. *eval-pf p* (*x* # *xs*))
| *eval-pf* (*Exist p*) *xs* = (∃ *x*. *eval-pf p* (*x* # *xs*))
| *eval-pf* (*Neg p*) *xs* = (¬ *eval-pf p xs*)

function *dfa-of-pf* :: *nat* ⇒ *pf* ⇒ *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* ⇒ *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 [λ(n, pf). count-forall pf, λ(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 ⟹
    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 (∃ k bs. eval-pf p (nat-of-bools bs # nats-of-boolss n bss) ∧ length bs =
length bss + k) =
    (∃ 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 ⇒ pf ⇒ 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* $\Longrightarrow$
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 (∃ *k bs*. *eval-pf p* (*nat-of-bools bs* # *nats-of-boolss n bss*) ∧ *length bs* = *length bss* + *k*) =
(∃ *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-len*2 *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, Proceed-*

ings, 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.