

Formalizing the Logic-Automaton Connection

Markus Reiter Stefan Berghofer

December 7, 2022

Abstract

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

Contents

1	General automata	1
2	BDDs	3
3	DFAs	5
3.1	Minimization	6
4	NFAs	13
5	Automata Constructions	16
5.1	Negation	16
5.2	Product Automaton	16
5.3	Transforming DFAs to NFAs	20
5.4	Transforming NFAs to DFAs	21
5.5	Quantifiers	24
5.6	Right Quotient	26
5.7	Diophantine Equations	28
5.8	Diophantine Inequations	34
6	Presburger Arithmetic	35
1	General automata	
	definition	

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

lemma *reach-nil*: $reach\ tr\ p\ []\ p \langle proof \rangle$

lemma *reach-snoc*: $reach\ tr\ p\ bs\ q \implies reach\ tr\ p\ (bs\ @\ [b])\ (tr\ q\ b)$
 $\langle proof \rangle$

lemma *reach-nil-iff*: $reach\ tr\ p\ []\ q = (p = q) \langle proof \rangle$

lemma *reach-snoc-iff*: $reach\ tr\ p\ (bs\ @\ [b])\ k = (\exists q. reach\ tr\ p\ bs\ q \wedge k = tr\ q\ b)$
 $\langle proof \rangle$

lemma *reach-induct* [*consumes* 1, *case-names* Nil *snoc*, *induct set*: *reach*]:

assumes $reach\ tr\ p\ w\ q$

and $P\ []\ p$

and $\bigwedge k\ x\ y. \llbracket reach\ tr\ p\ x\ k; P\ x\ k \rrbracket \implies P\ (x\ @\ [y])\ (tr\ k\ y)$

shows $P\ w\ q$

$\langle proof \rangle$

lemma *reach-trans*: $\llbracket reach\ tr\ p\ a\ r; reach\ tr\ r\ b\ q \rrbracket \implies reach\ tr\ p\ (a\ @\ b)\ q$
 $\langle proof \rangle$

lemma *reach-inj*: $\llbracket reach\ tr\ p\ a\ q; reach\ tr\ p\ a\ q' \rrbracket \implies q = q'$
 $\langle proof \rangle$

definition

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

locale *Automaton* =

fixes $trans :: 'a \Rightarrow 'b \Rightarrow 'a$

and $is-node :: 'a \Rightarrow bool$

and $is-alpha :: 'b \Rightarrow bool$

assumes $trans-is-node: \bigwedge q\ a. \llbracket is-node\ q; is-alpha\ a \rrbracket \implies is-node\ (trans\ q\ a)$

begin

lemma *steps-is-node*:

assumes $is-node\ q$

and $list-all\ is-alpha\ w$

shows $is-node\ (foldl\ trans\ q\ w)$

$\langle proof \rangle$

lemma *reach-is-node*: $\llbracket reach\ trans\ p\ w\ q; is-node\ p; list-all\ is-alpha\ w \rrbracket \implies is-node$
 q

$\langle proof \rangle$

end

2 BDDs

definition

is- α :: $\text{nat} \Rightarrow \text{bool list} \Rightarrow \text{bool}$ **where**
is- α $n = (\lambda w. \text{length } w = n)$

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

primrec bddh :: $\text{nat} \Rightarrow 'a \text{ bdd} \Rightarrow \text{bool}$

where

$\text{bddh } n \text{ (Leaf } x) = \text{True}$
 $| \text{bddh } n \text{ (Branch } l \ r) = (\text{case } n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } m \Rightarrow \text{bddh } m \ l \wedge \text{bddh } m \ r)$

lemma bddh-ge:

assumes $m \geq n$

assumes $\text{bddh } n \text{ bdd}$

shows $\text{bddh } m \text{ bdd}$

<proof>

abbreviation bdd-all $\equiv \text{pred-bdd}$

fun bdd-lookup :: 'a bdd $\Rightarrow \text{bool list} \Rightarrow 'a$

where

$\text{bdd-lookup (Leaf } x) \text{ bs} = x$
 $| \text{bdd-lookup (Branch } l \ r) \text{ (b\#bs)} = \text{bdd-lookup (if b then r else l) bs}$

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

<proof>

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

<proof>

lemma bdd-all-bdd-map:

assumes $\text{bdd-all } P \text{ bdd}$

and $\bigwedge a. P \ a \Longrightarrow Q \ (f \ a)$

shows $\text{bdd-all } Q \text{ (bdd-map } f \text{ bdd)}$

<proof>

lemma bddh-bdd-map:

shows $\text{bddh } n \text{ (bdd-map } f \text{ bdd)} = \text{bddh } n \text{ bdd}$

<proof>

lemma bdd-map-bdd-lookup:

assumes $\text{bddh (length } ws) \text{ bdd}$

shows $\text{bdd-lookup (bdd-map } f \text{ bdd) } ws = f \text{ (bdd-lookup bdd } ws)$

<proof>

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*)
⟨*proof*⟩

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

lemma *bdd-all-bdd-binop*:

assumes *bdd-all* *P* *bdd*

and *bdd-all* *Q* *bdd'*

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

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

⟨*proof*⟩

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

List.insert *x* (*List.insert* *x* *xs*) = *List.insert* *x* *xs*

⟨*proof*⟩

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

⟨*proof*⟩

lemma *add-leaves-bdd-all-eq*:

list-all *P* (*add-leaves* *tr* *xs*) \longleftrightarrow *bdd-all* *P* *tr* ∧ *list-all* *P* *xs*

⟨*proof*⟩

lemmas *add-leaves-bdd-all-eq'* =

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

lemma *add-leaves-mono*:

set *xs* \subseteq *set* *ys* \implies *set* (*add-leaves* *tr* *xs*) \subseteq *set* (*add-leaves* *tr* *ys*)

⟨*proof*⟩

lemma *add-leaves-binop-subset*:

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

3 DFAs

type-synonym *bddtable* = *nat bdd list*

type-synonym *astate* = *bool list*

type-synonym *dfa* = *bddtable* \times *astate*

definition

dfa-is-node :: *dfa* \Rightarrow *nat* \Rightarrow *bool* **where**
dfa-is-node *A* = ($\lambda q. q < length (fst A)$)

definition

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

definition

dfa-trans :: *dfa* \Rightarrow *nat* \Rightarrow *bool list* \Rightarrow *nat* **where**
dfa-trans *A* *q* *bs* \equiv *bdd-lookup* (*fst A* ! *q*) *bs*

definition

dfa-accepting :: *dfa* \Rightarrow *nat* \Rightarrow *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*)

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* \equiv *foldl* (*dfa-trans* *A*)

abbreviation *dfa-accepts* *A* \equiv *accepts* (*dfa-trans* *A*) (*dfa-accepting* *A*) 0

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

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

3.1 Minimization

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

where

$make\text{-}tr\ f\ 0\ i = []$
 $| make\text{-}tr\ f\ (Suc\ n)\ i = f\ i\ \# make\text{-}tr\ f\ n\ (Suc\ i)$

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

where

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

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

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

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

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

fun $check\text{-}eq :: nat\ bdd \Rightarrow nat\ bdd \Rightarrow bool\ list\ list \Rightarrow bool$ **where**

$check\text{-}eq\ (Leaf\ i)\ (Leaf\ j)\ T = (\neg\ tr\text{-}lookup\ T\ i\ j)\ |$
 $check\text{-}eq\ (Branch\ l\ r)\ (Leaf\ i)\ T = (check\text{-}eq\ l\ (Leaf\ i)\ T \wedge check\text{-}eq\ r\ (Leaf\ i)\ T)\ |$
 $check\text{-}eq\ (Leaf\ i)\ (Branch\ l\ r)\ T = (check\text{-}eq\ (Leaf\ i)\ l\ T \wedge check\text{-}eq\ (Leaf\ i)\ r\ T)\ |$
 $check\text{-}eq\ (Branch\ l1\ r1)\ (Branch\ l2\ r2)\ T = (check\text{-}eq\ l1\ l2\ T \wedge check\text{-}eq\ r1\ r2\ T)$

definition $iter :: dfa \Rightarrow bool\ list\ list \Rightarrow bool \times bool\ list\ list$ **where**

$iter = (\lambda bd, as)\ T. fold\text{-}map\text{-}idx\ (\lambda i. fold\text{-}map\text{-}idx\ (\lambda j\ c\ b. let\ b' = b \vee \neg\ check\text{-}eq\ (bd\ !\ i)\ (bd\ !\ j)\ T in\ (c \vee b \neq b',\ b'))\ 0)\ 1\ False\ T)$

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

$count\text{-}tr = foldl\ (foldl\ (\lambda y\ x. if\ x\ then\ y\ else\ Suc\ y))\ 0$

lemma $fold\text{-}map\text{-}idx\text{-}fst\text{-}snd\text{-}eq:$

assumes $f: \bigwedge i\ c\ x. fst\ (f\ i\ c\ x) = (c \vee x \neq snd\ (f\ i\ c\ x))$

shows $fst\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs) = (c \vee xs \neq snd\ (fold\text{-}map\text{-}idx\ f\ i\ c\ xs))$

⟨proof⟩

lemma *foldl-mono*:

assumes $f: \bigwedge x y y'. y < y' \implies f y x < f y' x$ **and** $y: y < y'$
shows $\text{foldl } f y xs < \text{foldl } f y' xs$ *<proof>*

lemma *fold-map-idx-count*:

assumes $f: \bigwedge i c x y. \text{fst } (f i c x) = (c \vee g y (\text{snd } (f i c x))) < (g y x :: \text{nat})$
and $f': \bigwedge i c x y. g y (\text{snd } (f i c x)) \leq g y x$
and $g: \bigwedge x y y'. y < y' \implies g y x < g y' x$
shows $\text{fst } (\text{fold-map-idx } f i c xs) =$
 $(c \vee \text{foldl } g y (\text{snd } (\text{fold-map-idx } f i c xs))) < \text{foldl } g y xs$
and $\text{foldl } g y (\text{snd } (\text{fold-map-idx } f i c xs)) \leq \text{foldl } g y xs$
<proof>

lemma *iter-count*:

assumes $\text{eq}: (b, T') = \text{iter } (bd, as) T$
and $b: b$
shows $\text{count-tr } T' < \text{count-tr } T$
<proof>

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

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

termination *<proof>*

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

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

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

lemma *fixpt-induct*:

assumes $H: \bigwedge M T. (\text{fst } (\text{iter } M T) \implies P M (\text{snd } (\text{iter } M T))) \implies P M T$
shows $P M T$
<proof>

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

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

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

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

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

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

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

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

lemma *fold-map-idx-nth*: $j < \text{length} \ xs \implies$
 $\text{snd} \ (\text{fold-map-idx } f \ i \ y \ xs) \ ! \ j = \text{snd} \ (f \ (i + j) \ (\text{fst} \ (\text{fold-map-idx } f \ i \ y \ (\text{take } j \ xs)))) \ (xs \ ! \ j))$
 ⟨proof⟩

lemma *init-tr-dist-nodes*:
 assumes *dfa-is-node* $M \ q$ and $p < q$
 shows $\text{tr-lookup} \ (\text{init-tr } M) \ q \ p = \text{dist-nodes } M \ 0 \ v \ p \ q$
 ⟨proof⟩

lemma *dist-nodes-suc*:
 $\text{dist-nodes } M \ (\text{Suc } n) \ v \ p \ q = (\exists \ bs. \text{is-alph } v \ bs \wedge \text{dist-nodes } M \ n \ v \ (\text{dfa-trans } M \ p \ bs) \ (\text{dfa-trans } M \ q \ bs))$
 ⟨proof⟩

lemma *bdd-lookup-append*:
 assumes *bddh* $n \ B$ and $\text{length} \ bs \geq n$
 shows $\text{bdd-lookup } B \ (bs \ @ \ w) = \text{bdd-lookup } B \ bs$
 ⟨proof⟩

lemma *bddh-exists*: $\exists n. \text{bddh } n \ B$
 ⟨proof⟩

lemma *check-eq-dist-nodes*:
 assumes $\forall p \ q. \text{dfa-is-node } M \ q \wedge \ p < q \implies \text{tr-lookup } T \ q \ p = (\exists n < m. \text{dist-nodes } M \ n \ v \ p \ q)$ and $m > 0$
 and *bdd-all* $(\text{dfa-is-node } M) \ l$ and *bdd-all* $(\text{dfa-is-node } M) \ r$
 shows $(\neg \text{check-eq } l \ r \ T) = (\exists bs. \text{bddh} \ (\text{length } bs) \ l \wedge \text{bddh} \ (\text{length } bs) \ r \wedge (\exists n < m. \text{dist-nodes } M \ n \ v \ (\text{bdd-lookup } l \ bs) \ (\text{bdd-lookup } r \ bs)))$
 ⟨proof⟩

lemma *iter-wf*: $\text{wf-tr } M \ T \implies \text{wf-tr } M \ (\text{snd} \ (\text{iter } M \ T))$
 ⟨proof⟩

lemma *fixpt-wf*: $\text{wf-tr } M \ T \implies \text{wf-tr } M \ (\text{fixpt } M \ T)$
 ⟨proof⟩

lemma *list-split*:
 assumes $n \leq \text{length} \ bss$
 shows $\exists b \ bs. \ bss = b \ @ \ bs \wedge \text{length } b = n$

<proof>

lemma *iter-dist-nodes*:

assumes *wf-tr M T*
and *wf-dfa M v*
and $\forall p q. \text{dfa-is-node } M q \wedge p < q \longrightarrow \text{tr-lookup } T q p = (\exists n < m. \text{dist-nodes } M n v p q)$ **and** $m > 0$
and *dfa-is-node M q and p < q*
shows $\text{tr-lookup } (\text{snd } (\text{iter } M T)) q p = (\exists n < \text{Suc } m. \text{dist-nodes } M n v p q)$
<proof>

lemma *fixpt-dist-nodes'*:

assumes *wf-tr M T and wf-dfa M v*
and $\forall p q. \text{dfa-is-node } M q \wedge p < q \longrightarrow \text{tr-lookup } T q p = (\exists n < m. \text{dist-nodes } M n v p q)$ **and** $m > 0$
and *dfa-is-node M q and p < q*
shows $\text{tr-lookup } (\text{fixpt } M T) q p = (\exists n. \text{dist-nodes } M n v p q)$
<proof>

lemma *fixpt-dist-nodes*:

assumes *wf-dfa M v*
and *dfa-is-node M p and dfa-is-node M q*
shows $\text{tr-lookup } (\text{fixpt } M (\text{init-tr } M)) p q = (\exists n. \text{dist-nodes } M n v p q)$
<proof>

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

where

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

lemma *mk-eqcl'-len*: $\text{length } (\text{mk-eqcl}' xs i j l T) = \text{length } xs$ *<proof>*

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

$\text{mk-eqcl } [] zs i T = ([], zs) |$
 $\text{mk-eqcl } (\text{None} \# xs) zs i T = (\text{let } (xs', zs') = \text{mk-eqcl } (\text{mk-eqcl}' xs i (\text{Suc } i) (\text{length } zs) T) (zs @ [i]) (\text{Suc } i) T \text{ in } (\text{length } zs \# xs', zs')) |$
 $\text{mk-eqcl } (\text{Some } l \# xs) zs i T = (\text{let } (xs', zs') = \text{mk-eqcl } xs zs (\text{Suc } i) T \text{ in } (l \# xs', zs'))$
<proof>

termination *<proof>*

lemma *mk-eqcl'-bound*:

assumes $\bigwedge x k. [x \in \text{set } xs; x = \text{Some } k] \Longrightarrow k < l$
and $x \in \text{set } (\text{mk-eqcl}' xs i j l T)$ **and** $x = \text{Some } k$
shows $k \leq l$
<proof>

lemma *mk-eqcl'-nth'*:

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

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

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

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

<proof>

lemma *mk-eqcl'-nth*:

assumes $\bigwedge i' j' k. \llbracket i' < \text{length } xs; j' < \text{length } xs; xs ! i' = \text{Some } k \rrbracket \implies (xs ! j' = \text{Some } k) = (\neg \text{tr-lookup } T (i' + jj) (j' + jj))$

and $\bigwedge a b c. \llbracket a \leq \text{length } T; b \leq \text{length } T; c \leq \text{length } T; \neg \text{tr-lookup } T a b; \neg \text{tr-lookup } T b c \rrbracket \implies \neg \text{tr-lookup } T a c$

and $\text{length } xs + jj = \text{length } T + 1$

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

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

and $ii < jj$

and $i < \text{length } xs$ **and** $\text{mk-eqcl}' xs ii jj l T ! i = \text{Some } m$

and $j < \text{length } xs$

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

<proof>

lemma *mk-eqcl'-Some*:

assumes $i < \text{length } xs$ **and** $xs ! i \neq \text{None}$

shows $\text{mk-eqcl}' xs ii j l T ! i = xs ! i$

<proof>

lemma *mk-eqcl'-Some2*:

assumes $i < \text{length } xs$

and $k < l$

shows $(\text{mk-eqcl}' xs ii j l T ! i = \text{Some } k) = (xs ! i = \text{Some } k)$

<proof>

lemma *mk-eqcl-fst-Some*:

assumes $i < \text{length } xs$ **and** $k < \text{length } zs$

shows $(\text{fst } (\text{mk-eqcl } xs zs ii T) ! i = k) = (xs ! i = \text{Some } k)$

<proof>

lemma *mk-eqcl-len-snd*:

$\text{length } zs \leq \text{length } (\text{snd } (\text{mk-eqcl } xs zs i T))$

<proof>

lemma *mk-eqcl-len-fst*:

$\text{length } (\text{fst } (\text{mk-eqcl } xs zs i T)) = \text{length } xs$

<proof>

lemma *mk-eqcl-set-snd*:

assumes $i \notin \text{set } zs$

and $j > i$

shows $i \notin \text{set } (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ j \ T))$
 <proof>

lemma *mk- eqcl -snd-mon:*

assumes $\bigwedge j1 \ j2. \llbracket j1 < j2; j2 < \text{length } zs \rrbracket \implies zs \ ! \ j1 < zs \ ! \ j2$
and $\bigwedge x. x \in \text{set } zs \implies x < i$
and $j1 < j2$ **and** $j2 < \text{length } (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ i \ T))$
shows $\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ i \ T) \ ! \ j1 < \text{snd } (\text{mk-} \text{eqcl } xs \ zs \ i \ T) \ ! \ j2$
 <proof>

lemma *mk- eqcl -snd-nth:*

assumes $i < \text{length } zs$
shows $\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ j \ T) \ ! \ i = zs \ ! \ i$
 <proof>

lemma *mk- eqcl -bound:*

assumes $\bigwedge x \ k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < \text{length } zs$
and $x \in \text{set } (\text{fst } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T))$
shows $x < \text{length } (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T))$
 <proof>

lemma *mk- eqcl -fst-snd:*

assumes $\bigwedge i. i < \text{length } zs \implies zs \ ! \ i < \text{length } xs + ii \wedge (zs \ ! \ i \geq ii \longrightarrow xs \ ! \ (zs \ ! \ i - ii) = \text{Some } i)$
and $\bigwedge j1 \ j2. \llbracket j1 < j2; j2 < \text{length } zs \rrbracket \implies zs \ ! \ j1 < zs \ ! \ j2$
and $\bigwedge z. z \in \text{set } zs \implies z < ii$
and $i < \text{length } (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T))$
and $\text{length } xs + ii \leq \text{length } T + 1$
shows $\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ i < \text{length } (\text{fst } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T)) + ii \wedge (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ i \geq ii \longrightarrow \text{fst } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ (\text{snd } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ i - ii) = i)$
 <proof>

lemma *mk- eqcl -fst-nth:*

assumes $\bigwedge i \ j \ k. \llbracket i < \text{length } xs; j < \text{length } xs; xs \ ! \ i = \text{Some } k \rrbracket \implies (xs \ ! \ j = \text{Some } k) = (\neg \text{tr-lookup } T \ (i + ii) \ (j + ii))$
and $\bigwedge a \ b \ c. \llbracket a \leq \text{length } T; b \leq \text{length } T; c \leq \text{length } T; \neg \text{tr-lookup } T \ a \ b; \neg \text{tr-lookup } T \ b \ c \rrbracket \implies \neg \text{tr-lookup } T \ a \ c$
and $\bigwedge x \ k. \llbracket x \in \text{set } xs; x = \text{Some } k \rrbracket \implies k < \text{length } zs$
and $\text{length } xs + ii = \text{length } T + 1$
and $i < \text{length } xs$ **and** $j < \text{length } xs$
shows $(\text{fst } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ i = \text{fst } (\text{mk-} \text{eqcl } xs \ zs \ ii \ T) \ ! \ j) = (\neg \text{tr-lookup } T \ (i + ii) \ (j + ii))$
 <proof>

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

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

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

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

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

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

shows $fst (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M)))$
 $! i < length (snd (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M))))$

(**is** $fst ?M ! i < length (snd ?M)$)

$\langle proof \rangle$

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

assumes $wf\text{-dfa } M v$

and $dfa\text{-is}\text{-node } M p$ **and** $dfa\text{-is}\text{-node } M q$

shows $(fst (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M))))$
 $! p = fst (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M))) ! q$
 $= eq\text{-nodes } M v p q$

(**is** $(fst ?M ! p = fst ?M ! q) = eq\text{-nodes } M v p q$)

$\langle proof \rangle$

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

assumes $i < length (snd (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M))))$

and $wf\text{-dfa } M v$

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** $snd ?M ! i < length (fst ?M) \wedge fst ?M ! (snd ?M ! i) = i$)

$\langle proof \rangle$

lemma $eq\text{-nodes}\text{-dfa}\text{-trans}$:

assumes $eq\text{-nodes } M v p q$

and $is\text{-alph } v bs$

shows $eq\text{-nodes } M v (dfa\text{-trans } M p bs) (dfa\text{-trans } M q bs)$

$\langle proof \rangle$

lemma $mk\text{-eqcl}\text{-fixpt}\text{-trans}$:

assumes $wf\text{-dfa } M v$

and $dfa\text{-is}\text{-node } M p$

and $is\text{-alph } v bs$

shows $dfa\text{-trans } (min\text{-dfa } M) (fst (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0$
 $(fixpt M (init\text{-tr } M))) ! p) bs =$

$fst (mk\text{-eqcl } (replicate (length (fst M)) None) [] 0 (fixpt M (init\text{-tr } M))) !$
 $(dfa\text{-trans } M p bs)$

(**is** $dfa\text{-trans } (min\text{-dfa } M) (fst ?M ! p) bs = fst ?M ! (dfa\text{-trans } M p bs)$)

$\langle proof \rangle$

lemma *mk-egcl-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-egcl (replicate (length (fst M)) None) [] 0 (fixpt M (init-tr M))) ! p) w =*
fst (mk-egcl (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)*
<proof>

lemma *mk-egcl-fixpt-startnode*:
assumes *length (fst M) > 0*
shows *length (snd (mk-egcl (replicate (length (fst M)) None) [] 0 (fixpt M (init-tr M)))) > 0* \wedge
fst (mk-egcl (replicate (length (fst M)) None) [] 0 (fixpt M (init-tr M))) ! 0 =
0 \wedge *snd (mk-egcl (replicate (length (fst M)) None) [] 0 (fixpt M (init-tr M))) ! 0 =*
0
(is *length (snd ?M) > 0* \wedge *fst ?M ! 0 = 0* \wedge *snd ?M ! 0 = 0*)
<proof>

lemma *min-dfa-wf*:
wf-dfa M v \implies *wf-dfa (min-dfa M) v*
<proof>

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>

4 NFAs

type-synonym *nbddtable = bool list bdd list*
type-synonym *nfa = nbddtable \times astate*

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

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

definition

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

fun

bv-or :: *bool list* \Rightarrow *bool list* \Rightarrow *bool list*

where

bv-or [] [] = [] |
bv-or (*x* # *xs*) (*y* # *ys*) = (*x* \vee *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* \vee *r* ! *i*)

<proof>

lemma *bv-or-length*:

assumes *length l* = *length r*
shows *length (bv-or l r)* = *length l*

<proof>

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* \cup *set-of-bv q*

<proof>

lemma *bv-or-is-node*: \llbracket *nfa-is-node A p*; *nfa-is-node A q* $\rrbracket \Longrightarrow$ *nfa-is-node A (bv-or p q)*

<proof>

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* \Rightarrow *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>

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')
 ⟨proof⟩

lemma *bdd-lookup-subsetbdd'*:
assumes length bdds = length q
and $\forall x \in \text{set } bdds. \text{bddh } (\text{length } ws) x$
and bddh (length ws) obdd
and $\bigwedge bs w. \llbracket bs \in \text{set } bdds; \text{length } w = \text{length } ws \rrbracket \implies \text{length } (\text{bdd-lookup } bs w) = c$
and $\bigwedge w. \text{length } w = \text{length } ws \implies \text{length } (\text{bdd-lookup } obdd w) = c$
and $a < c$
shows $\text{bdd-lookup } (\text{subsetbdd } bdds q obdd) ws ! a = ((\exists i < \text{length } q. q ! i \wedge \text{bdd-lookup } (bdds ! i) ws ! a) \vee \text{bdd-lookup } obdd ws ! a)$
 ⟨proof⟩

lemma *bdd-lookup-subsetbdd*:
assumes wf-nfa N (length ws)
and nfa-is-node N q
and $a < \text{length } (\text{fst } N)$
shows $\text{bdd-lookup } (\text{subsetbdd } (\text{fst } N) q (\text{nfa-emptybdd } (\text{length } q))) ws ! a = (\exists i < \text{length } q. q ! i \wedge \text{bdd-lookup } (\text{fst } N ! i) ws ! a)$
 ⟨proof⟩

definition

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

fun *nfa-accepting'* :: bool list \Rightarrow bool list \Rightarrow bool **where**
nfa-accepting' [] bs = False
 | *nfa-accepting'* as [] = False
 | *nfa-accepting'* (a # as) (b # bs) = (a \wedge b \vee *nfa-accepting'* as bs)

definition *nfa-accepting* :: nfa \Rightarrow bool list \Rightarrow bool **where**
nfa-accepting A = *nfa-accepting'* (snd A)

lemma *nfa-accepting'-set-of-bv*: *nfa-accepting'* l r = (set-of-bv l \cap set-of-bv r \neq {})
 ⟨proof⟩

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

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*>

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

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

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

lemma *nfa-startnode-is-node: wf-nfa* $A\ n \implies nfa-is-node\ A\ (nfa-startnode\ A)$
 <*proof*>

5 Automata Constructions

5.1 Negation

definition

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

lemma *negate-wf-dfa: wf-dfa* (*negate-dfa* A) $l = wf-dfa\ A\ l$
 <*proof*>

lemma *negate-negate-dfa: negate-dfa* (*negate-dfa* A) = A
 <*proof*>

lemma *dfa-accepts-negate:*

assumes *wf-dfa* $A\ n$
 and *list-all* (*is-alph* n) bss
 shows *dfa-accepts* (*negate-dfa* A) $bss = (\neg\ dfa-accepts\ A\ bss)$
 <*proof*>

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-formed1*: *wf-dfa* *A n*
and *well-formed2*: *wf-dfa* *B n*

sublocale *prod-DFS* < *DFS* *prod-succs* *A B* *prod-is-node* *A B* *prod-invariant* *A B*
prod-ins *prod-memb* *prod-empt* *A B*
⟨*proof*⟩

context *prod-DFS*

begin

lemma *prod-dfs-eq-rtrancl*: $prod-is-node\ A\ B\ x \implies prod-is-node\ A\ B\ y \implies prod-memb\ y\ (prod-dfs\ A\ B\ x) = ((x, y) \in (succsr\ (prod-succs\ A\ B))^*)$
<proof>

lemma *prod-dfs-bij*:

assumes $x: prod-is-node\ A\ B\ x$
shows $(fst\ (prod-dfs\ A\ B\ x) ! i ! j = Some\ k \wedge dfa-is-node\ A\ i \wedge dfa-is-node\ B\ j) = (k < length\ (snd\ (prod-dfs\ A\ B\ x)) \wedge (snd\ (prod-dfs\ A\ B\ x) ! k = (i, j)))$
<proof>

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 $fst\ (gen-dfs\ (prod-succs\ A\ B)\ prod-ins\ prod-memb\ z\ xs) ! i ! j = Some\ k$
<proof>

lemma *prod-dfs-start*:

$\llbracket dfa-is-node\ A\ i; dfa-is-node\ B\ j \rrbracket \implies fst\ (prod-dfs\ A\ B\ (i, j)) ! i ! j = Some\ 0$
<proof>

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>

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>

lemma *binop-wf-dfa*: $wf-dfa\ (binop-dfa\ f\ A\ B)\ n$
<proof>

theorem *binop-dfa-reachable*:

assumes *bss*: *list-all (is-alph n) bss*

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

<proof>

lemma *binop-dfa-steps*:

assumes *X*: *list-all (is-alph n) bs*

shows $\text{snd } (\text{binop-dfa } f \ A \ B) \ ! \ \text{dfa-steps } (\text{binop-dfa } f \ A \ B) \ 0 \ bs = f \ (\text{snd } A \ !$
 $\text{dfa-steps } A \ 0 \ bs) \ (\text{snd } B \ ! \ \text{dfa-steps } B \ 0 \ bs)$

(is *?as3* *! dfa-steps ?A 0 bs = ?rhs*)

<proof>

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>

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 $\text{dfa-accepts } (\text{binop-dfa } f \ A \ B) \ bss = f \ (\text{dfa-accepts } A \ bss) \ (\text{dfa-accepts } B$
 $bss)$

<proof>

definition

and-dfa :: *dfa* \Rightarrow *dfa* \Rightarrow *dfa* **where**

and-dfa = *binop-dfa* (\wedge)

lemma *and-wf-dfa*:

assumes *wf-dfa M n*

and *wf-dfa N n*

shows *wf-dfa (and-dfa M N) n*

<proof>

lemma *and-dfa-accepts*:

assumes *wf-dfa M n*

and *wf-dfa N n*

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

shows $\text{dfa-accepts } (\text{and-dfa } M \ N) \ bs = (\text{dfa-accepts } M \ bs \wedge \text{dfa-accepts } N \ bs)$

<proof>

definition

or-dfa :: *dfa* \Rightarrow *dfa* \Rightarrow *dfa* **where**

$or\text{-}dfa = binop\text{-}dfa (\vee)$

lemma *or-wf-dfa*:

assumes $wf\text{-}dfa\ M\ n$ **and** $wf\text{-}dfa\ N\ n$
shows $wf\text{-}dfa\ (or\text{-}dfa\ M\ N)\ n$
 $\langle proof \rangle$

lemma *or-dfa-accepts*:

assumes $wf\text{-}dfa\ M\ n$ **and** $wf\text{-}dfa\ N\ n$
and $list\text{-}all\ (is\text{-}alph\ n)\ bs$
shows $dfa\text{-}accepts\ (or\text{-}dfa\ M\ N)\ bs = (dfa\text{-}accepts\ M\ bs \vee dfa\text{-}accepts\ N\ bs)$
 $\langle proof \rangle$

definition

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

lemma *imp-wf-dfa*:

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

lemma *imp-dfa-accepts*:

assumes $wf\text{-}dfa\ M\ n$ **and** $wf\text{-}dfa\ N\ n$
and $list\text{-}all\ (is\text{-}alph\ n)\ bs$
shows $dfa\text{-}accepts\ (imp\text{-}dfa\ M\ N)\ bs = (dfa\text{-}accepts\ M\ bs \longrightarrow dfa\text{-}accepts\ N\ bs)$
 $\langle proof \rangle$

5.3 Transforming DFAs to NFAs

definition

$nfa\text{-}of\text{-}dfa :: dfa \Rightarrow nfa$ **where**
 $nfa\text{-}of\text{-}dfa = (\lambda(bdd, as). (map\ (bdd\text{-}map\ (\lambda q. (replicate\ (length\ bdd)\ False)[q:=True]))\ bdd, as))$

lemma *dfa2wf-nfa*:

assumes $wf\text{-}dfa\ M\ n$
shows $wf\text{-}nfa\ (nfa\text{-}of\text{-}dfa\ M)\ n$
 $\langle proof \rangle$

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

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

assumes $V: wf\text{-}dfa\ M\ l$
and $X: list\text{-}all\ (is\text{-}alph\ l)\ bss$
and $N: n1 = (replicate\ (length\ (fst\ M))\ False)[q:=True]$
and $Q: dfa\text{-}is\text{-}node\ M\ q$

and R : $nfa\text{-reach } (nfa\text{-of-dfa } M) \ n1 \ bss \ n2$
shows $\exists p. \ dfa\text{-reach } M \ q \ bss \ p \wedge \ n2 = (\text{replicate } (\text{length } (fst \ M)) \ False)[p:=True]$
 $\langle proof \rangle$

lemma $nfa\text{-of-dfa-reach}$:

assumes V : $wf\text{-dfa } M \ l$
and X : $list\text{-all } (is\text{-alph } l) \ bss$
and $N1$: $n1 = (\text{replicate } (\text{length } (fst \ M)) \ False)[q:=True]$
and $N2$: $n2 = (\text{replicate } (\text{length } (fst \ M)) \ False)[p:=True]$
and Q : $dfa\text{-is-node } M \ q$
shows $nfa\text{-reach } (nfa\text{-of-dfa } M) \ n1 \ bss \ n2 = dfa\text{-reach } M \ q \ bss \ p$
 $\langle proof \rangle$

lemma $nfa\text{-accepting-replicate}$:

assumes $q < \text{length } (fst \ N)$
and $\text{length } (snd \ N) = \text{length } (fst \ N)$
shows $nfa\text{-accepting } N \ ((\text{replicate } (\text{length } (fst \ N)) \ False)[q:=True]) = snd \ N \ ! \ q$
 $\langle proof \rangle$

lemma $nfa\text{-of-dfa-accepts}$:

assumes V : $wf\text{-dfa } A \ n$
and X : $list\text{-all } (is\text{-alph } n) \ bss$
shows $nfa\text{-accepts } (nfa\text{-of-dfa } A) \ bss = dfa\text{-accepts } A \ bss$
 $\langle proof \rangle$

5.4 Transforming NFAs to DFAs

fun

$bddinsert :: 'a \ bdd \Rightarrow \text{bool } list \Rightarrow 'a \Rightarrow 'a \ bdd$

where

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

lemma $bddh\text{-bddinsert}$:

assumes $bddh \ x \ b$
and $\text{length } w \geq x$
shows $bddh \ (\text{length } w) \ (bddinsert \ b \ w \ y)$
 $\langle proof \rangle$

lemma $bdd\text{-lookup-bddinsert}$:

assumes $bddh \ (\text{length } w) \ bd$
and $\text{length } w = \text{length } v$
shows $bdd\text{-lookup } (bddinsert \ bd \ w \ y) \ v = (\text{if } w = v \ \text{then } y \ \text{else } bdd\text{-lookup } bd \ v)$
 $\langle proof \rangle$

definition

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

definition

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

definition

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

definition

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

definition

subset-empt :: *nat option bdd* \times *bool list list* **where**
subset-empt = (*Leaf None*, \square)

definition

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

definition

det-nfa :: *nfa* \Rightarrow *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*}
 \langle *proof* \rangle

sublocale *subset-DFS* < *DFS* *subset-succs A nfa-is-node A subset-invariant A*
subset-ins subset-memb subset-empt
 \langle *proof* \rangle

context *subset-DFS*

begin

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

lemma *subset-dfs-bij*:

assumes *H1: nfa-is-node A q*

and *H2*: *nfa-is-node* *A* *q0*
shows $(bdd\text{-lookup } (fst \ (subset\text{-dfs } A \ q0)) \ q = \text{Some } v) = (v < length \ (snd \ (subset\text{-dfs } A \ q0)) \wedge (snd \ (subset\text{-dfs } A \ q0)) ! v = q)$
 $\langle proof \rangle$

lemma *subset-dfs-start*:
assumes *H*: *nfa-is-node* *A* *q0*
shows $bdd\text{-lookup } (fst \ (subset\text{-dfs } A \ q0)) \ q0 = \text{Some } 0$
 $\langle proof \rangle$

lemma *subset-dfs-is-node*:
assumes *nfa-is-node* *A* *q0*
shows $list\text{-all } (nfa\text{-is-node } A) \ (snd \ (subset\text{-dfs } A \ q0))$
 $\langle proof \rangle$

lemma *det-wf-nfa*:
shows $wf\text{-dfa } (det\text{-nfa } A) \ n$
 $\langle proof \rangle$

lemma *nfa-reach-rtrancl*:
assumes *nfa-is-node* *A* *i*
shows $(\exists \ bss. \ nfa\text{-reach } A \ i \ bss \ j \wedge list\text{-all } (is\text{-alph } n) \ bss) = ((i, j) \in (succsr \ (subset\text{-succs } A))^*)$
 $\langle proof \rangle$

lemma *nfa-reach-subset-memb*:
assumes *R*: *nfa-reach* *A* *q0* *bss* *q*
and *Q0*: *nfa-is-node* *A* *q0*
and *X*: $list\text{-all } (is\text{-alph } n) \ bss$
shows $subset\text{-memb } q \ (subset\text{-dfs } A \ q0)$
 $\langle proof \rangle$

lemma *det-nfa-reach'*:
fixes *bd* :: *nat option bdd* **and** *ls* :: *bool list list*
assumes $subset\text{-dfs } A \ (nfa\text{-startnode } A) = (bd, ls)$ (**is** *?subset-dfs = -*)
and $\exists \ bs. \ nfa\text{-reach } A \ (nfa\text{-startnode } A) \ bs \ q1 \wedge list\text{-all } (is\text{-alph } n) \ bs$
and $q1 = ls ! i$ **and** $q2 = ls ! j$ **and** $i < length \ ls$ **and** $j < length \ ls$
and $list\text{-all } (is\text{-alph } n) \ bss$
shows $nfa\text{-reach } A \ q1 \ bss \ q2 = (dfa\text{-reach } (det\text{-nfa } A) \ i \ bss \ j \wedge nfa\text{-is-node } A \ q2)$
 $(is \ - = (dfa\text{-reach } ?M \ i \ bss \ j \wedge -))$
 $\langle proof \rangle$

lemma *det-nfa-reach*:
fixes *bd* :: *nat option bdd* **and** *ls* :: *bool list list*
assumes S : $subset\text{-dfs } A \ (nfa\text{-startnode } A) = (bd, ls)$ (**is** *?subset-dfs = -*)
and *Q1*: $q1 = ls ! j$ **and** *J*: $j < length \ ls$
and *X*: $list\text{-all } (is\text{-alph } n) \ bss$
shows $nfa\text{-reach } A \ (nfa\text{-startnode } A) \ bss \ q1 = dfa\text{-reach } (det\text{-nfa } A) \ 0 \ bss \ j$
 $\langle proof \rangle$

lemma *det-nfa-accepts*:
assumes X : *list-all (is-alph n) w*
shows *dfa-accepts (det-nfa A) w = nfa-accepts A w*
<proof>

end

lemma *det-wf-nfa*:
assumes A : *wf-nfa A n*
shows *wf-dfa (det-nfa A) n*
<proof>

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>

5.5 Quantifiers

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

lemma *bddh-quantify-bdd*:
assumes *bddh (Suc n) bdd and v ≤ n*
shows *bddh n (quantify-bdd v bdd)*
<proof>

lemma *quantify-bdd-is-node*:
assumes *bdd-all (nfa-is-node N) bdd*
shows *bdd-all (nfa-is-node N) (quantify-bdd v bdd)*
<proof>

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

lemma *quantify-nfa-well-formed-aut*:
assumes *wf-nfa N (Suc n)*
and $v \leq n$
shows *wf-nfa (quantify-nfa v N) n*
<proof>

fun *insertl* :: $nat \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
insertl i a [] = [a]
| *insertl 0 a bs = a # bs*

| $\text{insertl } (\text{Suc } i) \ a \ (b \ \# \ bs) = b \ \# \ (\text{insertl } i \ a \ bs)$

lemma *insertl-len*:

$\text{length } (\text{insertl } n \ x \ vs) = \text{Suc } (\text{length } vs)$
<proof>

lemma *insertl-0-eq*: $\text{insertl } 0 \ x \ xs = x \ \# \ xs$

<proof>

lemma *bdd-lookup-quantify-bdd-set-of-bv*:

assumes $\text{length } w = n$
and $\text{bddh } (\text{Suc } n) \ bdd$
and $\text{bdd-all } (\text{nfa-is-node } N) \ bdd$
and $v \leq n$
shows $\text{set-of-bv } (\text{bdd-lookup } (\text{quantify-bdd } v \ bdd) \ w) = (\bigcup b. \text{set-of-bv } (\text{bdd-lookup } bdd \ (\text{insertl } v \ b \ w)))$
<proof>

lemma *subsetbdd-set-of-bv*:

assumes $\text{wf-nfa } N \ (\text{length } ws)$
and $\text{nfa-is-node } N \ q$
shows $\text{set-of-bv } (\text{bdd-lookup } (\text{subsetbdd } (\text{fst } N) \ q \ (\text{nfa-emptybdd } (\text{length } q))) \ ws) = (\bigcup i \in \text{set-of-bv } q. \text{set-of-bv } (\text{bdd-lookup } (\text{fst } N \ ! \ i) \ ws))$
(**is** $\text{set-of-bv } ?q = -$)
<proof>

lemma *nfa-trans-quantify-nfa*:

assumes $\text{wf-nfa } N \ (\text{Suc } n)$
and $v \leq n$
and $\text{is-alph } n \ w$
and $\text{nfa-is-node } N \ q$
shows $\text{set-of-bv } (\text{nfa-trans } (\text{quantify-nfa } v \ N) \ q \ w) = (\bigcup b. \text{set-of-bv } (\text{nfa-trans } N \ q \ (\text{insertl } v \ b \ w)))$
<proof>

fun *insertll* :: $\text{nat} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list list} \Rightarrow 'a \ \text{list list}$

where

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

lemma *insertll-len2*:

assumes $\text{list-all } (\text{is-alph } n) \ vs$
and $\text{length } x = \text{length } vs$
shows $\text{list-all } (\text{is-alph } (\text{Suc } n)) \ (\text{insertll } k \ x \ vs)$
<proof>

lemma *insertll-append*:

assumes $\text{length } xs = \text{length } vs$
shows $\text{insertll } k \ (xs \ @ \ [x]) \ (vs \ @ \ [v]) = \text{insertll } k \ xs \ vs \ @ \ [\text{insertl } k \ x \ v]$

<proof>

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

<proof>

lemma *nfa-steps-quantify-nfa*:

assumes *wf-nfa* $N \ (\text{Suc } n)$

and *list-all* $(\text{is-alph } n) \ w$

and *nfa-is-node* $N \ q$

and $v \leq n$

shows *set-of-bv* $(\text{nfa-steps } (\text{quantify-nfa } v \ N) \ q \ w) = (\bigcup xs \in \{x. \text{length } x = \text{length } w\}. \text{set-of-bv } (\text{nfa-steps } N \ q \ (\text{insertll } v \ xs \ w)))$

<proof>

lemma *nfa-accepts-quantify-nfa*:

assumes *wf-nfa* $A \ (\text{Suc } n)$

and $i \leq n$

and *list-all* $(\text{is-alph } n) \ bss$

shows *nfa-accepts* $(\text{quantify-nfa } i \ A) \ bss = (\exists bs. \text{nfa-accepts } A \ (\text{insertll } i \ bs \ bss) \wedge \text{length } bs = \text{length } bss)$

<proof>

5.6 Right Quotient

definition

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

definition

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

definition

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

definition

rquot-memb :: $\text{nat} \Rightarrow \text{bool list} \Rightarrow \text{bool}$ **where**
rquot-memb = $(\lambda x \ l. l \ ! \ x)$

definition

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

definition

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

definition

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

lemma *zeros-is-alpha*: *list-all (is- α ph v) (zeros n v)*
 <proof>

lemma *zeros-rone*: *zeros (Suc n) v = zeros n v @ zeros 1 v*
 <proof>

lemma *zeros-len*: *length (zeros n v) = n*
 <proof>

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

primrec *map-index* :: (*'a* \Rightarrow *nat* \Rightarrow *'b*) \Rightarrow *'a list* \Rightarrow *nat* \Rightarrow *'b list*
where

map-index *f [] n* = []
 | *map-index* *f (x#xs) n* = *f x n # map-index f xs (Suc n)*

lemma *map-index-len*:
length (map-index f ls n) = length ls
 <proof>

lemma *map-index-nth*:
assumes *i < length l*
shows *map-index f l n ! i = f (l ! i) (n + i)*
 <proof>

definition

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

lemma *rquot-well-formed-aut*:
assumes *wf-dfa M n*
shows *wf-dfa (rquot M n) n*
 <proof>

lemma *rquot-node*:
dfa-is-node (rquot M n) q = dfa-is-node M q
 <proof>

lemma *rquot-steps*:
dfa-steps (rquot M n) x w = dfa-steps M x w
 <proof>

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-empty A
  <proof>

context rquot-DFS
begin

lemma rquot-dfs-invariant:
  assumes dfa-is-node A x
  shows rquot-invariant A (rquot-dfs A n x)
  <proof>

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>

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>

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>

```

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 (λx. x * n) xs)
  <proof>

lemma eval-dioph-add-map:
  eval-dioph ks (map f xs) + eval-dioph ks (map g xs) =
  eval-dioph ks (map (λx. f x + g x) (xs::nat list))

```

$\langle proof \rangle$

lemma *eval-dioph-div-mult*:

$eval_dioph\ ks\ (map\ (\lambda x. x\ div\ n)\ xs) * int\ n +$
 $eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ n)\ xs) = eval_dioph\ ks\ xs$
 $\langle proof \rangle$

lemma *eval-dioph-mod*:

$eval_dioph\ ks\ xs\ mod\ int\ n = eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ n)\ xs)\ mod\ int\ n$
 $\langle proof \rangle$

lemma *eval-dioph-div-mod*:

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

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

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

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

$\langle proof \rangle$

lemma *zmult-div-aux1*:

assumes $b: b \neq 0$
shows $(a - a\ mod\ b)\ div\ b = (a::int)\ div\ b$
 $\langle proof \rangle$

lemma *zmult-div-aux2*:

assumes $b: b \neq 0$
shows $((a::int) - a\ mod\ b)\ mod\ b = 0$
 $\langle proof \rangle$

lemma *div-abs-eq*:

assumes $mod: (a::int)\ mod\ b = 0$
and $b: 0 < b$
shows $|a\ div\ b| = |a|\ div\ b$
 $\langle proof \rangle$

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

$\langle proof \rangle$

lemma *dioph-rhs-bound*:

$|l - eval_dioph\ ks\ (map\ (\lambda x. x\ mod\ 2)\ xs)\ div\ 2| \leq max\ |l|\ (\sum\ k \leftarrow ks.\ |k|)$
 $\langle proof \rangle$

lemma *dioph-rhs-invariant*:

assumes m : $|m| \leq \max |l| (\sum k \leftarrow ks. |k|)$

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

<proof>

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

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

shows *finite* S

<proof>

primrec *mk-nat-vecs* :: $\text{nat} \Rightarrow \text{nat list list}$ **where**

$\text{mk-nat-vecs } 0 = []$

| $\text{mk-nat-vecs } (\text{Suc } n) =$

$(\text{let } yss = \text{mk-nat-vecs } n$

$\text{in } \text{map } (\text{Cons } 0) yss @ \text{map } (\text{Cons } 1) yss)$

lemma *mk-nat-vecs-bound*: $\forall xs \in \text{set } (\text{mk-nat-vecs } n). \forall x \in \text{set } xs. x < 2$

<proof>

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

$= xs$

<proof>

definition

dioph-succs n ks $m = \text{List.map-filter } (\lambda xs.$

$\text{if } \text{eval-dioph } ks xs \bmod 2 = m \bmod 2$

$\text{then } \text{Some } ((m - \text{eval-dioph } ks xs) \text{ div } 2)$

$\text{else } \text{None}) (\text{mk-nat-vecs } n)$

definition

dioph-is-node :: $\text{int list} \Rightarrow \text{int} \Rightarrow \text{int} \Rightarrow \text{bool}$ **where**

$\text{dioph-is-node } ks l m = (|m| \leq \max |l| (\sum k \leftarrow ks. |k|))$

definition

dioph-invariant :: $\text{int list} \Rightarrow \text{int} \Rightarrow \text{nat option list} \times \text{int list} \Rightarrow \text{bool}$ **where**

$\text{dioph-invariant } ks l = (\lambda (is, js). \text{length } is = \text{nat } (2 * \max |l| (\sum k \leftarrow ks. |k|) + 1))$

definition

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

definition

dioph-memb :: $\text{int} \Rightarrow \text{nat option list} \times \text{int list} \Rightarrow \text{bool}$ **where**

$\text{dioph-memb } m = (\lambda (is, js). is ! \text{int-encode } m \neq \text{None})$

definition

dioph-empt :: $\text{int list} \Rightarrow \text{int} \Rightarrow \text{nat option list} \times \text{int list}$ **where**

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

lemma *int-encode-bound*: $dioph\text{-}is\text{-}node\ ks\ l\ m \implies$
 $int\text{-}encode\ m < nat\ (2 * max\ |l|\ (\sum\ k \leftarrow ks.\ |k|) + 1)$
 $\langle proof \rangle$

interpretation *dioph-dfs*: $DFS\ dioph\text{-}succs\ n\ ks\ dioph\text{-}is\text{-}node\ ks\ l$
 $dioph\text{-}invariant\ ks\ l\ dioph\text{-}ins\ dioph\text{-}memb\ dioph\text{-}empt\ ks\ l$
 $\langle proof \rangle$

definition
 $dioph\text{-}dfs\ n\ ks\ l = gen\text{-}dfs\ (dioph\text{-}succs\ n\ ks)\ dioph\text{-}ins\ dioph\text{-}memb\ (dioph\text{-}empt\ ks\ l)\ [l]$

primrec *make-bdd* :: $(nat\ list \Rightarrow 'a) \Rightarrow nat \Rightarrow nat\ list \Rightarrow 'a\ bdd$
where

$make\text{-}bdd\ f\ 0\ xs = Leaf\ (f\ xs)$
 $| make\text{-}bdd\ f\ (Suc\ n)\ xs = Branch\ (make\text{-}bdd\ f\ n\ (xs\ @\ [0]))\ (make\text{-}bdd\ f\ n\ (xs\ @\ [1]))$

definition

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

abbreviation *input* $nat\text{-}of\text{-}bool :: bool \Rightarrow nat$

where

$nat\text{-}of\text{-}bool \equiv of\text{-}bool$

lemma *nat-of-bool-bound*: $nat\text{-}of\text{-}bool\ b < 2$
 $\langle proof \rangle$

lemma *nat-of-bool-mk-nat-vecs*:

$length\ bs = n \implies map\ nat\text{-}of\text{-}bool\ bs \in set\ (mk\text{-}nat\text{-}vecs\ n)$
 $\langle proof \rangle$

lemma *bdd-lookup-make-bdd*:

$length\ bs = n \implies bdd\text{-}lookup\ (make\text{-}bdd\ f\ n\ xs)\ bs = f\ (xs\ @\ map\ nat\text{-}of\text{-}bool\ bs)$
 $\langle proof \rangle$

primrec *nat-of-bools* :: $bool\ list \Rightarrow nat$

where

$nat\text{-}of\text{-}bools\ [] = 0$
 $| nat\text{-}of\text{-}bools\ (b \# bs) = nat\text{-}of\text{-}bool\ b + 2 * nat\text{-}of\text{-}bools\ bs$

primrec *nats-of-boolss* :: *nat* \Rightarrow *bool list list* \Rightarrow *nat list*

where

Nil: *nats-of-boolss* *n* [] = *replicate* *n* 0

| *Cons*: *nats-of-boolss* *n* (*bs* # *bss*) =

map ($\lambda(b, x). \text{nat-of-bool } b + 2 * x$) (*zip* *bs* (*nats-of-boolss* *n* *bss*))

lemma *nats-of-boolss-length*:

list-all (*is-alph* *n*) *bss* \Longrightarrow *length* (*nats-of-boolss* *n* *bss*) = *n*

<proof>

lemma *nats-of-boolss-mod2*:

assumes *bs*: *length* *bs* = *n* **and** *bss*: *list-all* (*is-alph* *n*) *bss*

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

<proof>

lemma *nats-of-boolss-div2*:

assumes *bs*: *length* *bs* = *n* **and** *bss*: *list-all* (*is-alph* *n*) *bss*

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

<proof>

lemma *zip-insertl*: *length* *xs* = *length* *ys* \Longrightarrow

zip (*insertl* *n* *x* *xs*) (*insertl* *n* *y* *ys*) = *insertl* *n* (*x*, *y*) (*zip* *xs* *ys*)

<proof>

lemma *map-insertl*: *map* *f* (*insertl* *i* *x* *xs*) = *insertl* *i* (*f* *x*) (*map* *f* *xs*)

<proof>

lemma *insertl-replicate*: *m* \leq *n* \Longrightarrow

insertl *m* *x* (*replicate* *n* *x*) = *x* # *replicate* *n* *x*

<proof>

lemma *nats-of-boolss-insertll*:

list-all (*is-alph* *n*) *bss* \Longrightarrow *length* *bs* = *length* *bss* \Longrightarrow *i* \leq *n* \Longrightarrow

nats-of-boolss (*Suc* *n*) (*insertll* *i* *bs* *bss*) = *insertl* *i* (*nat-of-bools* *bs*) (*nats-of-boolss* *n* *bss*)

<proof>

lemma *zip-replicate-map*: *length* *xs* = *n* \Longrightarrow *zip* (*replicate* *n* *x*) *xs* = *map* (*Pair* *x*) *xs*

<proof>

lemma *zip-replicate-mapr*: *length* *xs* = *n* \Longrightarrow *zip* *xs* (*replicate* *n* *x*) = *map* ($\lambda y. (y, x)$) *xs*

<proof>

lemma *zip-assoc*: *map* *f* (*zip* *xs* (*zip* *ys* *zs*)) = *map* ($\lambda((x, y), z). f(x, (y, z))$) (*zip* (*zip* *xs* *ys*) *zs*)

<proof>

lemma *nats-of-boolss-append*:
 $list-all (is-alph n) bss \implies list-all (is-alph n) bss' \implies$
 $nats-of-boolss n (bss @ bss') =$
 $map (\lambda(x, y). x + 2 ^ length bss * y) (zip (nats-of-boolss n bss) (nats-of-boolss$
 $n bss'))$
 $\langle proof \rangle$

lemma *nats-of-boolss-zeros*: $nats-of-boolss n (zeros m n) = replicate n 0$
 $\langle proof \rangle$

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

fun *bools-of-nat* :: $nat \Rightarrow nat \Rightarrow bool list$
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*: $k \leq length (bools-of-nat k n)$
 $\langle proof \rangle$

lemma *nat-of-bool-mod-eq*: $nat-of-bool (n mod 2 = 1) = n mod 2$
 $\langle proof \rangle$

lemma *bools-of-nat-inverse*: $nat-of-bools (bools-of-nat k n) = n$
 $\langle proof \rangle$

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

lemma *eval-dioph-replicate-0*: $eval-dioph ks (replicate n 0) = 0$
 $\langle proof \rangle$

lemma *dioph-dfs-bij*:
 $(fst (dioph-dfs n ks l) ! int-encode i = Some k \wedge dioph-is-node ks l i) =$
 $(k < length (snd (dioph-dfs n ks l)) \wedge (snd (dioph-dfs n ks l) ! k = i))$
 $\langle proof \rangle$

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 $fst (gen-dfs (dioph-succs n ks) dioph-ins dioph-memb z xs) ! i = Some k$
 $\langle proof \rangle$

lemma *dioph-dfs-start*:
 $fst (dioph-dfs n ks l) ! int-encode l = Some 0$
 $\langle proof \rangle$

lemma *eq-dfa-error*: \neg *dfa-accepting* (*eq-dfa* *n ks l*) (*dfa-steps* (*eq-dfa* *n ks l*) (*length* (*snd* (*dioph-dfs* *n ks l*))) *bss*)
 ⟨*proof*⟩

lemma *eq-dfa-accepting*:
 (*l*, *m*) ∈ (*succsr* (*dioph-succs* *n ks*))* \implies *list-all* (*is-alph* *n*) *bss* \implies
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*⟩

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*)
 ⟨*proof*⟩

lemma *bddh-make-bdd*: *bddh* *n* (*make-bdd* *f n xs*)
 ⟨*proof*⟩

lemma *bdd-all-make-bdd*: *bdd-all* *P* (*make-bdd* *f n xs*) = (\forall *ys* ∈ *set* (*mk-nat-vecs* *n*). *P* (*f* (*xs* @ *ys*)))
 ⟨*proof*⟩

lemma *eq-wf-dfa*: *wf-dfa* (*eq-dfa* *n ks l*) *n*
 ⟨*proof*⟩

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*⟩

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*:
 $(fst (dioph-ineq-dfs n ks l) ! int-encode i = Some k \wedge dioph-is-node ks l i) =$
 $(k < length (snd (dioph-ineq-dfs n ks l)) \wedge (snd (dioph-ineq-dfs n ks l) ! k = i))$
 $\langle proof \rangle$

lemma *dioph-ineq-dfs-mono*:
assumes z : *dioph-invariant ks l z*
and xs : *list-all (dioph-is-node ks l) xs*
and H : $fst z ! i = Some k$
shows $fst (gen-dfs (dioph-ineq-sucss n ks) dioph-ins dioph-memb z xs) ! i = Some k$
 $\langle proof \rangle$

lemma *dioph-ineq-dfs-start*:
 $fst (dioph-ineq-dfs n ks l) ! int-encode l = Some 0$
 $\langle proof \rangle$

lemma *ineq-dfa-accepting*:
 $(l, m) \in (succsr (dioph-ineq-sucss n ks))^* \implies list-all (is-alph n) bss \implies$
 $dfa-accepting (ineq-dfa n ks l) (dfa-steps (ineq-dfa n ks l) (the (fst (dioph-ineq-dfs$
 $n ks l) ! int-encode m)) bss) =$
 $(eval-dioph ks (nats-of-boolss n bss) \leq m)$
 $\langle proof \rangle$

lemma *ineq-dfa-accepts*:
assumes bss : *list-all (is-alph n) bss*
shows $dfa-accepts (ineq-dfa n ks l) bss = (eval-dioph ks (nats-of-boolss n bss) \leq$
 $l)$
 $\langle proof \rangle$

lemma *ineq-wf-dfa*: $wf-dfa (ineq-dfa n ks l) n$
 $\langle proof \rangle$

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 $eval-pf :: pf \Rightarrow passign \Rightarrow bool$
where

$eval\text{-}pf (Eq\ ks\ l)\ xs = (eval\text{-}dioph\ ks\ xs = l)$
 $| eval\text{-}pf (Le\ ks\ l)\ xs = (eval\text{-}dioph\ ks\ xs \leq l)$
 $| eval\text{-}pf (And\ p\ q)\ xs = (eval\text{-}pf\ p\ xs \wedge eval\text{-}pf\ q\ xs)$
 $| eval\text{-}pf (Or\ p\ q)\ xs = (eval\text{-}pf\ p\ xs \vee eval\text{-}pf\ q\ xs)$
 $| eval\text{-}pf (Imp\ p\ q)\ xs = (eval\text{-}pf\ p\ xs \longrightarrow eval\text{-}pf\ q\ xs)$
 $| eval\text{-}pf (Forall\ p)\ xs = (\forall x. eval\text{-}pf\ p\ (x \# xs))$
 $| eval\text{-}pf (Exist\ p)\ xs = (\exists x. eval\text{-}pf\ p\ (x \# xs))$
 $| eval\text{-}pf (Neg\ p)\ xs = (\neg eval\text{-}pf\ p\ xs)$

function $dfa\text{-}of\text{-}pf :: nat \Rightarrow pf \Rightarrow dfa$

where

$Eq: dfa\text{-}of\text{-}pf\ n\ (Eq\ ks\ l) = eq\text{-}dfa\ n\ ks\ l$
 $| Le: dfa\text{-}of\text{-}pf\ n\ (Le\ ks\ l) = ineq\text{-}dfa\ n\ ks\ l$
 $| And: dfa\text{-}of\text{-}pf\ n\ (And\ p\ q) = and\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
 $| Or: dfa\text{-}of\text{-}pf\ n\ (Or\ p\ q) = or\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
 $| Imp: dfa\text{-}of\text{-}pf\ n\ (Imp\ p\ q) = imp\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)\ (dfa\text{-}of\text{-}pf\ n\ q)$
 $| Exist: dfa\text{-}of\text{-}pf\ n\ (Exist\ p) = rquot\ (det\text{-}nfa\ (quantify\text{-}nfa\ 0\ (nfa\text{-}of\text{-}dfa\ (dfa\text{-}of\text{-}pf\ (Suc\ n)\ p))))\ n$
 $| Forall: dfa\text{-}of\text{-}pf\ n\ (Forall\ p) = dfa\text{-}of\text{-}pf\ n\ (Neg\ (Exist\ (Neg\ p)))$
 $| Neg: dfa\text{-}of\text{-}pf\ n\ (Neg\ p) = negate\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)$
 $\langle proof \rangle$

Auxiliary measure function for termination proof

primrec $count\text{-}forall :: pf \Rightarrow nat$

where

$count\text{-}forall\ (Eq\ ks\ l) = 0$
 $| count\text{-}forall\ (Le\ ks\ l) = 0$
 $| count\text{-}forall\ (And\ p\ q) = count\text{-}forall\ p + count\text{-}forall\ q$
 $| count\text{-}forall\ (Or\ p\ q) = count\text{-}forall\ p + count\text{-}forall\ q$
 $| count\text{-}forall\ (Imp\ p\ q) = count\text{-}forall\ p + count\text{-}forall\ q$
 $| count\text{-}forall\ (Exist\ p) = count\text{-}forall\ p$
 $| count\text{-}forall\ (Forall\ p) = 1 + count\text{-}forall\ p$
 $| count\text{-}forall\ (Neg\ p) = count\text{-}forall\ p$

termination $dfa\text{-}of\text{-}pf$

$\langle proof \rangle$

lemmas $dfa\text{-}of\text{-}pf\text{-}induct =$

$dfa\text{-}of\text{-}pf.\text{induct}\ [case\text{-}names\ Eq\ Le\ And\ Or\ Imp\ Exist\ Forall\ Neg]$

lemma $dfa\text{-}of\text{-}pf\text{-}well\text{-}formed: wf\text{-}dfa\ (dfa\text{-}of\text{-}pf\ n\ p)\ n$

$\langle proof \rangle$

lemma $dfa\text{-}of\text{-}pf\text{-}correctness:$

$list\text{-}all\ (is\text{-}alph\ n)\ bss \Longrightarrow$

$dfa\text{-}accepts\ (dfa\text{-}of\text{-}pf\ n\ p)\ bss = eval\text{-}pf\ p\ (nats\text{-}of\text{-}boolss\ n\ bss)$

$\langle proof \rangle$

The same with minimization after quantification.

function $dfa\text{-of}\text{-}pf' :: nat \Rightarrow pf \Rightarrow dfa$
where
 $dfa\text{-of}\text{-}pf' n (Eq\ ks\ l) = eq\text{-}dfa\ n\ ks\ l$
 $| dfa\text{-of}\text{-}pf' n (Le\ ks\ l) = ineq\text{-}dfa\ n\ ks\ l$
 $| dfa\text{-of}\text{-}pf' n (And\ p\ q) = and\text{-}dfa\ (dfa\text{-of}\text{-}pf' n\ p)\ (dfa\text{-of}\text{-}pf' n\ q)$
 $| dfa\text{-of}\text{-}pf' n (Or\ p\ q) = or\text{-}dfa\ (dfa\text{-of}\text{-}pf' n\ p)\ (dfa\text{-of}\text{-}pf' n\ q)$
 $| dfa\text{-of}\text{-}pf' n (Imp\ p\ q) = imp\text{-}dfa\ (dfa\text{-of}\text{-}pf' n\ p)\ (dfa\text{-of}\text{-}pf' n\ q)$
 $| dfa\text{-of}\text{-}pf' n (Exist\ p) = min\text{-}dfa\ (rquot\ (det\text{-}nfa\ (quantify\text{-}nfa\ 0\ (nfa\text{-of}\text{-}dfa\ (dfa\text{-of}\text{-}pf'\ (Suc\ n)\ p))))\ n)$
 $| dfa\text{-of}\text{-}pf' n (Forall\ p) = dfa\text{-of}\text{-}pf' n\ (Neg\ (Exist\ (Neg\ p)))$
 $| dfa\text{-of}\text{-}pf' n (Neg\ p) = negate\text{-}dfa\ (dfa\text{-of}\text{-}pf' n\ p)$
 $\langle proof \rangle$

termination $dfa\text{-of}\text{-}pf'$
 $\langle proof \rangle$

lemmas $dfa\text{-of}\text{-}pf'\text{-}induct =$
 $dfa\text{-of}\text{-}pf'.induct\ [case\text{-}names\ Eq\ Le\ And\ Or\ Imp\ Exist\ Forall\ Neg]$

lemma $dfa\text{-of}\text{-}pf'\text{-}well\text{-}formed: wf\text{-}dfa\ (dfa\text{-of}\text{-}pf' n\ p)\ n$
 $\langle proof \rangle$

lemma $dfa\text{-of}\text{-}pf'\text{-}correctness:$
 $list\text{-}all\ (is\text{-}alph\ n)\ bss \implies$
 $dfa\text{-accepts}\ (dfa\text{-of}\text{-}pf' n\ p)\ bss = eval\text{-}pf\ p\ (nats\text{-of}\text{-}boolss\ n\ bss)$
 $\langle proof \rangle$

References

- [1] S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 147–163. Springer-Verlag, 2009.
- [2] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.
- [3] N. Klarlund. Mona & Fido: The logic-automaton connection in practice. In M. Nielsen and W. Thomas, editors, *Computer Science Logic, 11th International Workshop, CSL '97, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, 1998.