Formalizing the Logic-Automaton Connection

Markus Reiter Stefan Berghofer

April 18, 2024

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* ⟨*proof*⟩

lemma *reach-snoc*: *reach tr p bs q* ⟹ *reach tr p* (*bs* @ [*b*]) (*tr q b*)
⟨*proof*⟩

lemma *reach-nil-iff*: *reach tr p* [] *q* = (*p* = *q*) ⟨*proof*⟩

lemma *reach-snoc-iff*: *reach tr p* (*bs* @ [*b*]) *k* = (∃ *q*. *reach tr p bs q* ∧ *k* = *tr q b*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *reach-trans*: ⟦*reach tr p a r*; *reach tr r b q*⟧ ⟹ *reach tr p* (*a* @ *b*) *q*
⟨*proof*⟩

lemma *reach-inj*: ⟦*reach tr p a q*; *reach tr p a q'*⟧ ⟹ *q* = *q'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reach-is-node*: ⟦*reach trans p w q*; *is-node p*; *list-all is-alpha w*⟧ ⟹ *is-node q*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bdd-all-bdd-lookup-iff*: *bddh n bdd* ⟹ *bdd-all P bdd* = (∀ *ws*. *length ws* = *n* ⟶ *P* (*bdd-lookup bdd ws*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bddh-bdd-map*:
shows *bddh n* (*bdd-map f bdd*) = *bddh n bdd*
⟨*proof*⟩

lemma *bdd-map-bdd-lookup*:
assumes *bddh* (*length ws*) *bdd*
shows *bdd-lookup* (*bdd-map f bdd*) *ws* = *f* (*bdd-lookup bdd ws*)
⟨*proof*⟩

fun *bdd-binop* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ bdd \Rightarrow 'b\ bdd \Rightarrow 'c\ bdd$
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*: ⟦*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*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *insert-list-idemp*[*simp*]:
List.insert x (*List.insert x xs*) = *List.insert x xs*
⟨*proof*⟩

primrec *add-leaves* :: $'a\ bdd \Rightarrow 'a\ list \Rightarrow 'a\ list$
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*)
⟨*proof*⟩

lemma *add-leaves-bdd-all-eq*:
list-all P (*add-leaves tr xs*) ⟷ *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 ⊆ *set ys* ⟹ *set* (*add-leaves tr xs*) ⊆ *set* (*add-leaves tr ys*)
⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *foldl-mono*:
  assumes *f*: $\bigwedge x\ y\ y'.\ y < y' \Longrightarrow f\ y\ x < f\ y'\ x$ **and** *y*: $y < y'$
  shows *foldl f y xs* $<$ *foldl f y' xs* ⟨*proof*⟩

lemma *fold-map-idx-count*:
  assumes *f*: $\bigwedge i\ c\ x\ y.$ *fst* (*f i c x*) = (*c* ∨ *g y* (*snd* (*f i c x*)) $<$ (*g y x*::*nat*))
  and *f'*: $\bigwedge i\ c\ x\ y.\ g\ y\ (snd\ (f\ i\ c\ x)) \leq g\ y\ x$
  and *g*: $\bigwedge x\ y\ y'.\ y < y' \Longrightarrow g\ y\ x < g\ y'\ x$
  shows *fst* (*fold-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*⟩

lemma *iter-count*:
  assumes *eq*: (*b*, *T'*) = *iter* (*bd*, *as*) *T*
  and *b*: *b*
  shows *count-tr T'* $<$ *count-tr T*
⟨*proof*⟩

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*)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *fixpt-True*[*simp*]: *fst* (*iter M T*) ⟹ *fixpt M T* = *fixpt M* (*snd* (*iter M T*))
  ⟨*proof*⟩

lemma *fixpt-False*[*simp*]: ¬ (*fst* (*iter M T*)) ⟹ *fixpt M T* = *T*
  ⟨*proof*⟩

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

lemma *fixpt-induct*:
  assumes *H*: $\bigwedge M\ T.$ (*fst* (*iter M T*) ⟹ *P M* (*snd* (*iter M T*))) ⟹ *P M T*
  shows *P M T*
⟨*proof*⟩

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*
  ⟨*proof*⟩

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

lemma *init-tr-wf*: *wf-tr M* (*init-tr M*)
⟨*proof*⟩

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

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))
⟨*proof*⟩

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*⟩

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

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

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

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

lemma *iter-wf*: *wf-tr M T* $\Longrightarrow$ *wf-tr M* (*snd* (*iter M T*))
⟨*proof*⟩

lemma *fixpt-wf*: *wf-tr M T* $\Longrightarrow$ *wf-tr M* (*fixpt M T*)
⟨*proof*⟩

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

⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩

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'*))
⟨*proof*⟩
termination ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *mk-eqcl'-nth'*:
assumes $\bigwedge x\ k.\ [\![x \in set\ xs;\ x = Some\ k]\!] \Longrightarrow k < l$
and $\bigwedge i'.\ [\![i' < length\ xs;\ \neg\ tr\text{-}lookup\ T\ (i' + j)\ i]\!] \Longrightarrow xs\ !\ i' = None$
and $i < j$ **and** $j' < length\ xs$
shows $(mk\text{-}eqcl'\ xs\ i\ j\ l\ T\ !\ j' = Some\ l) = (\neg\ tr\text{-}lookup\ T\ (j' + j)\ i)$
⟨*proof*⟩

lemma *mk-eqcl'-nth*:
assumes $\bigwedge i'\ j'\ k.\ [\![i' < length\ xs;\ j' < length\ xs;\ xs\ !\ i' = Some\ k]\!] \Longrightarrow (xs\ !\ j' = Some\ k) = (\neg\ tr\text{-}lookup\ T\ (i' + jj)\ (j' + jj))$
and $\bigwedge a\ b\ c.\ [\![a \leq length\ T;\ b \leq length\ T;\ c \leq length\ T;\ \neg\ tr\text{-}lookup\ T\ a\ b;\ \neg\ tr\text{-}lookup\ T\ b\ c]\!] \Longrightarrow \neg\ tr\text{-}lookup\ T\ a\ c$
and $length\ xs + jj = length\ T + 1$
and $\bigwedge x\ k.\ [\![x \in set\ xs;\ x = Some\ k]\!] \Longrightarrow k < l$
and $\bigwedge i'.\ [\![i' < length\ xs;\ \neg\ tr\text{-}lookup\ T\ (i' + jj)\ ii]\!] \Longrightarrow xs\ !\ i' = None$
and $ii < jj$
and $i < length\ xs$ **and** $mk\text{-}eqcl'\ xs\ ii\ jj\ l\ T\ !\ i = Some\ m$
and $j < length\ xs$
shows $(mk\text{-}eqcl'\ xs\ ii\ jj\ l\ T\ !\ j = Some\ m) = (\neg\ tr\text{-}lookup\ T\ (i + jj)\ (j + jj))$
⟨*proof*⟩

lemma *mk-eqcl'-Some*:
assumes $i < length\ xs$ **and** $xs\ !\ i \neq None$
shows $mk\text{-}eqcl'\ xs\ ii\ j\ l\ T\ !\ i = xs\ !\ i$
⟨*proof*⟩

lemma *mk-eqcl'-Some2*:
assumes $i < length\ xs$
and $k < l$
shows $(mk\text{-}eqcl'\ xs\ ii\ j\ l\ T\ !\ i = Some\ k) = (xs\ !\ i = Some\ k)$
⟨*proof*⟩

lemma *mk-eqcl-fst-Some*:
assumes $i < length\ xs$ **and** $k < length\ zs$
shows $(fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i = k) = (xs\ !\ i = Some\ k)$
⟨*proof*⟩

lemma *mk-eqcl-len-snd*:
$length\ zs \leq length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T))$
⟨*proof*⟩

lemma *mk-eqcl-len-fst*:
$length\ (fst\ (mk\text{-}eqcl\ xs\ zs\ i\ T)) = length\ xs$
⟨*proof*⟩

lemma *mk-eqcl-set-snd*:
assumes $i \notin set\ zs$
and $j > i$

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

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 $snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T)\ !\ j1 < snd\ (mk\text{-}eqcl\ xs\ zs\ i\ T)\ !\ j2$
⟨*proof*⟩

lemma *mk-eqcl-snd-nth*:
assumes $i < length\ zs$
shows $snd\ (mk\text{-}eqcl\ xs\ zs\ j\ T)\ !\ i = zs\ !\ i$
⟨*proof*⟩

lemma *mk-eqcl-bound*:
assumes $\bigwedge x\ k.\ [\![x \in set\ xs;\ x = Some\ k]\!] \Longrightarrow k < length\ zs$
and $x \in set\ (fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T))$
shows $x < length\ (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T))$
⟨*proof*⟩

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 $snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i < length\ (fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)) + ii \wedge (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i \geq ii \longrightarrow\ fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ (snd\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i - ii) = i)$
⟨*proof*⟩

lemma *mk-eqcl-fst-nth*:
assumes $\bigwedge i\ j\ k.\ [\![i < length\ xs;\ j < length\ xs;\ xs\ !\ i = Some\ k]\!] \Longrightarrow (xs\ !\ j = Some\ k) = (\neg\ tr\text{-}lookup\ T\ (i + ii)\ (j + ii))$
and $\bigwedge a\ b\ c.\ [\![a \leq length\ T;\ b \leq length\ T;\ c \leq length\ T;\ \neg\ tr\text{-}lookup\ T\ a\ b;\ \neg\ tr\text{-}lookup\ T\ b\ c]\!] \Longrightarrow \neg\ tr\text{-}lookup\ T\ a\ c$
and $\bigwedge x\ k.\ [\![x \in set\ xs;\ x = Some\ k]\!] \Longrightarrow k < length\ zs$
and $length\ xs + ii = length\ T + 1$
and $i < length\ xs$ **and** $j < length\ xs$
shows $(fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ i = fst\ (mk\text{-}eqcl\ xs\ zs\ ii\ T)\ !\ j) = (\neg\ tr\text{-}lookup\ T\ (i + ii)\ (j + ii))$
⟨*proof*⟩

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

definition *eq-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*⟩

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*⟩

lemma *mk-eqcl-fixpt-fst-snd-nth*:
assumes *i* < *length* (*snd* (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*))))
and *wf-dfa M v*
shows *snd* (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*)))
! *i* < *length* (*fst* (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*)))) ∧
fst (*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*))) ! (*snd*
(*mk-eqcl* (*replicate* (*length* (*fst M*)) *None*) [] 0 (*fixpt M* (*init-tr M*))) ! *i*) = *i*
(**is** *snd ?M* ! *i* < *length* (*fst ?M*) ∧ *fst ?M* ! (*snd ?M* ! *i*) = *i*)
⟨*proof*⟩

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*⟩

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*⟩

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*))
⟨*proof*⟩

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*⟩

lemma *min-dfa-wf*:
wf-dfa M v ⟹ *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* × *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*)
⟨*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* ∪ *set-of-bv q*
⟨*proof*⟩

lemma *bv-or-is-node*: ⟦*nfa-is-node A p*; *nfa-is-node A q*⟧ ⟹ *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* ⇒ *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 ∀ *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*)
⟨*proof*⟩

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*⟩

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

fun *nfa-accepting'* :: *bool list* ⇒ *bool list* ⇒ *bool* **where**
nfa-accepting' [] *bs* = *False*
| *nfa-accepting' as* [] = *False*
| *nfa-accepting'* (*a* # *as*) (*b* # *bs*) = (*a* ∧ *b* ∨ *nfa-accepting' as bs*)
definition *nfa-accepting* :: *nfa* ⇒ *bool list* ⇒ *bool* **where**
nfa-accepting A = *nfa-accepting'* (*snd A*)

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

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

definition
nfa-startnode :: *nfa* ⇒ *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* ≡ *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*)
  ⟨*proof*⟩

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*
  ⟨*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* = (¬ *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-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*
⟨*proof*⟩

context *prod-DFS*

begin

lemma *prod-dfs-eq-rtrancl*: *prod-is-node A B x* $\Longrightarrow$ *prod-is-node A B y* $\Longrightarrow$
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*:
⟦*dfa-is-node A i*; *dfa-is-node B j*⟧ $\Longrightarrow$ *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** *i*1: *dfa-is-node A i*1 **and** *i*2: *dfa-is-node B i*2
and *j*1: *dfa-is-node A j*1 **and** *j*2: *dfa-is-node B j*2
and *i*: *fst* (*prod-dfs A B x*) ! *i*1 ! *i*2 = *Some k*
and *j*: *fst* (*prod-dfs A B x*) ! *j*1 ! *j*2 = *Some k*
shows (*i*1, *i*2) = (*j*1, *j*2)
⟨*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 *s*1: *dfa-is-node A s*1 **and** *s*2: *dfa-is-node B s*2
and *k*: *fst* (*prod-dfs A B* (*s*1, *s*2)) ! *i* ! *j* = *Some k*
obtains *k*′
where *fst* (*prod-dfs A B* (*s*1, *s*2)) !
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* (*s*1, *s*2)))
⟨*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 (∃ *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*⟩

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*⟩

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 *dfa-accepts* (*binop-dfa f A B*) *bss* = *f* (*dfa-accepts A bss*) (*dfa-accepts B bss*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *replicate-upd-inj*: ⟦*q* < *n*; (*replicate n False*)[*q*:=*True*] = (*replicate n False*)[*p*:=*True*]⟧
⟹ (*q* = *p*) (**is** ⟦- ;*?lhs* = *?rhs*⟧ ⟹ -)
⟨*proof*⟩

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*⟩

lemma *nfa-of-dfa-reach*:
assumes V: *wf-dfa* M l
and X: *list-all* (*is-alph* l) *bss*
and $N1$: $n1 =$ (*replicate* (*length* (*fst* M)) *False*)[q:=*True*]
and $N2$: $n2 =$ (*replicate* (*length* (*fst* M)) *False*)[p:=*True*]
and Q: *dfa-is-node* M q
shows *nfa-reach* (*nfa-of-dfa* M) $n1$ *bss* $n2$ = *dfa-reach* M q *bss* p
⟨*proof*⟩

lemma *nfa-accepting-replicate*:
assumes $q <$ *length* (*fst* N)
and *length* (*snd* N) = *length* (*fst* N)
shows *nfa-accepting* N ((*replicate* (*length* (*fst* N)) *False*)[q:=*True*]) = *snd* N ! q
⟨*proof*⟩

lemma *nfa-of-dfa-accepts*:
assumes V: *wf-dfa* A n
and X: *list-all* (*is-alph* n) *bss*
shows *nfa-accepts* (*nfa-of-dfa* A) *bss* = *dfa-accepts* A *bss*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*}
⟨*proof*⟩

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

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*⟩

lemma *subset-dfs-start*:
assumes *H*: *nfa-is-node A q*0
shows *bdd-lookup* (*fst* (*subset-dfs A q*0)) *q*0 = *Some* 0
⟨*proof*⟩

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*⟩

lemma *det-wf-nfa*:
shows *wf-dfa* (*det-nfa A*) *n*
⟨*proof*⟩

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*⟩

lemma *nfa-reach-subset-memb*:
assumes *R*: *nfa-reach A q*0 *bss q*
and *Q*0: *nfa-is-node A q*0
and *X*: *list-all* (*is-alph n*) *bss*
shows *subset-memb q* (*subset-dfs A q*0)
⟨*proof*⟩

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*⟩

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 *Q*1: *q*1 = *ls* ! *j* **and** *J*: *j* < *length ls*
and *X*: *list-all* (*is-alph n*) *bss*
shows *nfa-reach A* (*nfa-startnode A*) *bss q*1 = *dfa-reach* (*det-nfa A*) 0 *bss j*
⟨*proof*⟩

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* ⇒ *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*)
⟨*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* ⇒ *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* ≤ *n*
shows *wf-nfa* (*quantify-nfa v N*) *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *insertl*-0-*eq*: *insertl* 0 *x xs* = *x* # *xs*
⟨*proof*⟩

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* ≤ *n*
shows *set-of-bv* (*bdd-lookup* (*quantify-bdd v bdd*) *w*) = (⋃ *b*. *set-of-bv* (*bdd-lookup bdd* (*insertl v b w*)))
⟨*proof*⟩

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*⟩

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*⟩

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-len*2:
assumes *list-all* (*is-alph n*) *vs*
and *length x* = *length vs*
shows *list-all* (*is-alph* (*Suc n*)) (*insertll k x vs*)
⟨*proof*⟩

lemma *insertll-append*:
assumes *length xs* = *length vs*
shows *insertll k* (*xs* @ [*x*]) (*vs* @ [*v*]) = *insertll k xs vs* @ [*insertl k x v*]

⟨*proof*⟩

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

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

lemma *nfa-accepts-quantify-nfa*:
assumes *wf-nfa A* (*Suc n*)
and $i \leq n$
and *list-all* (*is-alph n*) *bss*
shows *nfa-accepts* (*quantify-nfa i A*) *bss* = $(\exists\, bs.$ *nfa-accepts A* (*insertll i bs bss*) $\wedge$ *length bs* = *length bss*)
⟨*proof*⟩

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*)
⟨*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*: (∃ *n*. *dfa-reach M x* (*zeros n v*) *y*) = ((*x*,*y*) ∈ (*succsr* (*rquot-succs M v*))*)
⟨*proof*⟩

primrec *map-index* :: ('*a* ⇒ *nat* ⇒ '*b*) ⇒ '*a list* ⇒ *nat* ⇒ '*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* ⇒ *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*
⟨*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-empt 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* (λ*x*. *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* (λ*x*. *f x* + *g x*) (*xs*::*nat list*))

⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

lemma *eval-dioph-ineq-div-mod*:
(*eval-dioph ks xs* ≤ *l*) =
(*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*⟩

lemma *sum-list-abs-ge-*0: (0::*int*) ≤ *sum-list* (*map abs ks*)
⟨*proof*⟩

lemma *zmult-div-aux*1:
assumes *b*: *b* ≠ 0
shows (*a* − *a mod b*) *div b* = (*a*::*int*) *div b*
⟨*proof*⟩

lemma *zmult-div-aux*2:
assumes *b*: *b* ≠ 0
shows ((*a*::*int*) − *a mod b*) *mod b* = 0
⟨*proof*⟩

lemma *div-abs-eq*:
assumes *mod*: (*a*::*int*) *mod b* = 0
and *b*: 0 < *b*
shows |*a div b*| = |*a*| *div b*
⟨*proof*⟩

lemma *add-div-trivial*: 0 ≤ *c* ⟹ *c* < *b* ⟹ ((*a*::*int*) ∗ *b* + *c*) *div b* = *a*
⟨*proof*⟩

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

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*⟩

lemma *bounded-int-set-is-finite*:
assumes *S*: $\forall (i::int){\in}S.\ |i| < j$
shows *finite S*
⟨*proof*⟩

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*: $\forall\, xs{\in}set\ (mk\text{-}nat\text{-}vecs\ n).\ \forall\, x{\in}set\ xs.\ x < 2$
⟨*proof*⟩

lemma *mk-nat-vecs-mod-eq*: *xs* ∈ *set* (*mk-nat-vecs n*) ⟹ *map* (λ*x*. *x mod* 2) *xs* = *xs*
⟨*proof*⟩

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

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

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*| $(\sum k{\leftarrow}ks.\ |k|)$ + 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)
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

lemma *nat-of-bool-mk-nat-vecs*:
length bs = *n* ⟹ *map nat-of-bool bs* ∈ *set* (*mk-nat-vecs n*)
⟨*proof*⟩

lemma *bdd-lookup-make-bdd*:
length bs = *n* ⟹ *bdd-lookup* (*make-bdd f n xs*) *bs* = *f* (*xs* @ *map nat-of-bool bs*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *zip-insertl*: *length xs* = *length ys* ⟹
  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* ≤ *n* ⟹
  insertl m x (*replicate n x*) = *x* # *replicate n x*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *zip-replicate-map*: *length xs* = *n* ⟹ *zip* (*replicate n x*) *xs* = *map* (*Pair x*) *xs*
  ⟨*proof*⟩

lemma *zip-replicate-mapr*: *length xs* = *n* ⟹ *zip xs* (*replicate n x*) = *map* (λ*y*. (*y*, *x*)) *xs*
  ⟨*proof*⟩

lemma *zip-assoc*: *map f* (*zip xs* (*zip ys zs*)) = *map* (λ((*x*, *y*), *z*). *f* (*x*, (*y*, *z*))) (*zip* (*zip xs ys*) *zs*)
  ⟨*proof*⟩

lemma *nats-of-boolss-append*:
list-all (*is-alph n*) *bss* $\Longrightarrow$ *list-all* (*is-alph n*) *bss'* $\Longrightarrow$
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'*))
⟨*proof*⟩

lemma *nats-of-boolss-zeros*: *nats-of-boolss n* (*zeros m n*) = *replicate n* 0
⟨*proof*⟩

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* ≤ *length* (*bools-of-nat k n*)
⟨*proof*⟩

lemma *nat-of-bool-mod-eq*: *nat-of-bool* (*n mod* 2 = 1) = *n mod* 2
⟨*proof*⟩

lemma *bools-of-nat-inverse*: *nat-of-bools* (*bools-of-nat k n*) = *n*
⟨*proof*⟩

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

lemma *eval-dioph-replicate*-0: *eval-dioph ks* (*replicate n* 0) = 0
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *dioph-dfs-start*:
fst (*dioph-dfs n ks l*) ! *int-encode l* = *Some* 0
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*) = (∀ *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* ∧ *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*⟩

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*
⟨*proof*⟩

lemma *dioph-ineq-dfs-start*:
fst (*dioph-ineq-dfs n ks l*) ! *int-encode l* = *Some* 0
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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)*
⟨*proof*⟩

termination *dfa-of-pf'*
⟨*proof*⟩

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*⟩

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*⟩

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.