A Formalization of Tree Automaton, (Anchord) Ground Tree Transducers, and Regular Relations*

Alexander Lochmann Bertram Felgenhauer
Christian Sternagel René Thiemann Thomas Sternagel

March 17, 2025

Abstract

Tree automata have good closure properties and therefore a commonly used to prove/disprove properties. This formalization contains among other things the proofs of many closure properties of tree automata (anchored) ground tree transducers and regular relations. Additionally it includes the well known pumping lemma and a lifting of the Myhill Nerode theorem for regular languages to tree languages.

We want to mention the existence of a tree automata APF-entry developed by Peter Lammich. His work is based on epsilon free top-down tree automata, while this entry builds on bottom-up tree auotamta with epsilon transitions. Moreover our formalization relies on the Collections Framework also by Peter Lammich [4] to obtain efficient code. All proven constructions of the closure properties are exportable using the Isabelle/HOL code generation facilities.

Contents

*Supported by FWF (Austrian Science Fund) projects P30301 and Y757.

1 Introduction

Tree automata characterize a computable subset of term languages which are called regular tree languages. These languages are closed under union, intersection, and complement. Due to their nice closure properties tree automata techniques are frequently used to prove/disprove properties.

As an example consider the field of rewriting. Dauchet and Tison showed that the theory of ground rewrite systems is decidable [2]. As another example, Kucherov et.al. proved that the regularity of the normal forms induced by a rewrite system is decidable [3].

In this formalization we also consider (anchored) ground tree transducers ((A)GTTs) and regular relations. The first allows to reason about relations on regular tree languages and the latter to reason about tuples of arbitrary size over regular tree languages. We distinguish them as they have different closure properties. While (anchored) ground tree transducers are closed under transitivity, regular relations are not. Additional information about these constructions and their closure properties can be found in [6].

This APF-entry provides a formalization of the general tree automata theory, GTTs, and regular relations. Moreover it contains a newly developed theory on the topic of AGTTs (construction is equivalent to the definition of Rec_2 in TATA [1, Chapter 3]) and how they are related to regular GTTs.

We want to mention the existence of a tree automata APF-entry developed by Peter Lammich [5]. The main reason for developing a new tree automata theory instead of working on top of his work was the underlying tree automata definition. Whereas our formalization defines bottom-up tree automaton with epsilon transitions, Peter Lammichs defines top-down tree automaton without epsilon transitions. These definitions do not differ in expressibility (i.e. a language is recognized by a bottom-up tree automaton if and only if it is recognized by a top-down tree automaton), however the use of epsilon transitions simplifies many constructions.

2 Preliminaries

theory *Term-Context*
  imports *First-Order-Terms.Term*
    First-Order-Terms.Subterm-and-Context
    Polynomial-Factorization.Missing-List
begin

2.1 Additional functionality on *Term.term* and *ctxt*

2.1.1 Positions

type-synonym *pos* = *nat list*
context
  notes *conj-cong* [*fundef-cong*]
begin

fun *poss* :: ('*f*, '*v*) *term* ⇒ *pos set* **where**
  poss (*Var x*) = {[]}
| *poss* (*Fun f ss*) = {[]} ∪ {*i* # *p* | *i p*. *i* < *length ss* ∧ *p* ∈ *poss* (*ss* ! *i*)}
end

fun *hole-pos* **where**
  hole-pos □ = []
| *hole-pos* (*More f ss D ts*) = *length ss* # *hole-pos D*

definition *position-less-eq* (**infixl** ‹≤$_p$› 67) **where**
  p ≤$_p$ *q* ⟷ (∃ *r*. *p* @ *r* = *q*)

abbreviation *position-less* (**infixl** ‹<$_p$› 67) **where**
  p <$_p$ *q* ≡ *p* ≠ *q* ∧ *p* ≤$_p$ *q*

definition *position-par* (**infixl** ‹⊥› 67) **where**
  p ⊥ *q* ⟷ ¬ (*p* ≤$_p$ *q*) ∧ ¬ (*q* ≤$_p$ *p*)

fun *remove-prefix* **where**
  remove-prefix (*x* # *xs*) (*y* # *ys*) = (*if x* = *y then remove-prefix xs ys else None*)
| *remove-prefix* [] *ys* = *Some ys*
| *remove-prefix xs* [] = *None*

definition *pos-diff* (**infixl** ‹−$_p$› 67) **where**
  p −$_p$ *q* = *the* (*remove-prefix q p*)

fun *subt-at* :: ('*f*, '*v*) *term* ⇒ *pos* ⇒ ('*f*, '*v*) *term* (**infixl** ‹|'-› 67) **where**
  s |- [] = *s*
| *Fun f ss* |- (*i* # *p*) = (*ss* ! *i*) |- *p*
| *Var x* |- - = *undefined*

fun *ctxt-at-pos* **where**
  ctxt-at-pos s [] = □

| *ctxt-at-pos* (*Fun f ss*) (*i # p*) = *More f* (*take i ss*) (*ctxt-at-pos* (*ss ! i*) *p*) (*drop* (*Suc i*) *ss*)
| *ctxt-at-pos* (*Var x*) - = *undefined*

fun *replace-term-at* (‹-[- ← -]› [*1000*, *0*, *0*] *1000*) **where**
replace-term-at s [] *t* = *t*
| *replace-term-at* (*Var x*) *ps t* = (*Var x*)
| *replace-term-at* (*Fun f ts*) (*i # ps*) *t* =
(*if i* < *length ts then Fun f* (*ts*[*i*:=(*replace-term-at* (*ts ! i*) *ps t*)]) *else Fun f ts*)

fun *fun-at* :: ('*f*, '*v*) *term* ⇒ *pos* ⇒ ('*f* + '*v*) *option* **where**
fun-at (*Var x*) [] = *Some* (*Inr x*)
| *fun-at* (*Fun f ts*) [] = *Some* (*Inl f*)
| *fun-at* (*Fun f ts*) (*i # p*) = (*if i* < *length ts then fun-at* (*ts ! i*) *p else None*)
| *fun-at* - - = *None*

2.1.2 Computing the signature

fun *funas-term* **where**
funas-term (*Var x*) = {}
| *funas-term* (*Fun f ts*) = *insert* (*f*, *length ts*) (⋃ (*set* (*map funas-term ts*)))

fun *funas-ctxt* **where**
funas-ctxt □ = {}
| *funas-ctxt* (*More f ss C ts*) = (⋃ (*set* (*map funas-term ss*))) ∪
insert (*f*, *Suc* (*length ss* + *length ts*)) (*funas-ctxt C*) ∪ (⋃ (*set* (*map funas-term ts*)))

2.1.3 Groundness

fun *ground* **where**
ground (*Var x*) = *False*
| *ground* (*Fun f ts*) = (∀ *t* ∈ *set ts*. *ground t*)

fun *ground-ctxt* **where**
ground-ctxt □ ⟷ *True*
| *ground-ctxt* (*More f ss C ts*) ⟷ (∀ *t* ∈ *set ss*. *ground t*) ∧ *ground-ctxt C* ∧ (∀ *t* ∈ *set ts*. *ground t*)

2.1.4 Depth

fun *depth* **where**
depth (*Var x*) = *0*
| *depth* (*Fun f* []) = *0*
| *depth* (*Fun f ts*) = *Suc* (*Max* (*depth* ‘ *set ts*))

2.1.5 Type conversion

We require a function which adapts the type of variables of a term, so that states of the automaton and variables in the term language can be chosen

independently.

abbreviation *map-funs-term f* ≡ *map-term f* (λ *x*. *x*)
abbreviation *map-both f* ≡ *map-prod f f*

definition *adapt-vars* :: ('*f*, '*q*) *term* ⇒ ('*f*,'*v*)*term* **where**
[*code del*]: *adapt-vars* ≡ *map-vars-term* (λ-. *undefined*)

abbreviation *map-vars-ctxt f* ≡ *map-ctxt* (λ*x*. *x*) *f*
definition *adapt-vars-ctxt* :: ('*f*,'*q*)*ctxt* ⇒ ('*f*,'*v*)*ctxt* **where**
[*code del*]: *adapt-vars-ctxt* = *map-vars-ctxt* (λ-. *undefined*)

2.2 Properties of *pos*

lemma *position-less-eq-induct* [*consumes 1*]:
assumes *p* ≤*p* *q* **and** ⋀ *p*. *P p p*
and ⋀ *p q r*. *p* ≤*p* *q* ⟹ *P p q* ⟹ *P p* (*q* @ *r*)
shows *P p q* **using** *assms*
proof (*induct p arbitrary*: *q*)
case *Nil* **then show** *?case*
by (*metis append-Nil position-less-eq-def*)
next
case (*Cons a p*)
then show *?case*
by (*metis append-Nil2 position-less-eq-def*)
qed

We show the correspondence between the function *remove-prefix* and the order on positions (≤*p*). Moreover how it can be used to compute the difference of positions.

lemma *remove-prefix-Nil* [*simp*]:
remove-prefix xs xs = *Some* []
by (*induct xs*) *auto*

lemma *remove-prefix-Some*:
assumes *remove-prefix xs ys* = *Some zs*
shows *ys* = *xs* @ *zs* **using** *assms*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs*)
show *?case* **using** *Cons*(*1*)[*of tl ys*] *Cons*(*2*)
by (*cases ys*) (*auto split*: *if-splits*)
qed *auto*

lemma *remove-prefix-append*:
remove-prefix xs (*xs* @ *ys*) = *Some ys*
by (*induct xs*) *auto*

lemma *remove-prefix-iff*:
remove-prefix xs ys = *Some zs* ⟷ *ys* = *xs* @ *zs*
using *remove-prefix-Some remove-prefix-append*

by *blast*

lemma *position-less-eq-remove-prefix*:
$p \leq_p q \Longrightarrow$ *remove-prefix* $p\ q \neq$ *None*
by (*induct rule*: *position-less-eq-induct*) (*auto simp*: *remove-prefix-iff*)

Simplification rules on $(\leq_p)$, $(-_p)$, and $(\perp)$.

lemma *position-less-refl* [*simp*]: $p \leq_p p$
by (*auto simp*: *position-less-eq-def*)

lemma *position-less-eq-Cons* [*simp*]:
$(i \# ps) \leq_p (j \# qs) \longleftrightarrow i = j \wedge ps \leq_p qs$
by (*auto simp*: *position-less-eq-def*)

lemma *position-less-Nil-is-bot* [*simp*]: $[] \leq_p p$
by (*auto simp*: *position-less-eq-def*)

lemma *position-less-Nil-is-bot2* [*simp*]: $p \leq_p [] \longleftrightarrow p = []$
by (*auto simp*: *position-less-eq-def*)

lemma *position-diff-Nil* [*simp*]: $q -_p [] = q$
by (*auto simp*: *pos-diff-def*)

lemma *position-diff-Cons* [*simp*]:
$(i \# ps) -_p (i \# qs) = ps -_p qs$
by (*auto simp*: *pos-diff-def*)

lemma *Nil-not-par* [*simp*]:
$[] \perp p \longleftrightarrow$ *False*
$p \perp [] \longleftrightarrow$ *False*
by (*auto simp*: *position-par-def*)

lemma *par-not-refl* [*simp*]: $p \perp p \longleftrightarrow$ *False*
by (*auto simp*: *position-par-def*)

lemma *par-Cons-iff*:
$(i \# ps) \perp (j \# qs) \longleftrightarrow (i \neq j \vee ps \perp qs)$
by (*auto simp*: *position-par-def*)

Simplification rules on *poss*.

lemma *Nil-in-poss* [*simp*]: $[] \in$ *poss* t
by (*cases* t) *auto*

lemma *poss-Cons* [*simp*]:
$i \# p \in$ *poss* $t \Longrightarrow [i] \in$ *poss* t
by (*cases* t) *auto*

lemma *poss-Cons-poss*:
$i \# q \in$ *poss* $t \longleftrightarrow i <$ *length* (*args* t) $\wedge\ q \in$ *poss* (*args* t ! i)

by (*cases t*) *auto*

lemma *poss-append-poss*:
p @ *q* ∈ *poss t* ⟷ *p* ∈ *poss t* ∧ *q* ∈ *poss* (*t* |- *p*)
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
from *Cons*[*of args t* ! *i*] **show** *?case*
by (*cases t*) *auto*
qed *auto*

Simplification rules on *hole-pos*

lemma *hole-pos-map-vars* [*simp*]:
hole-pos (*map-vars-ctxt f C*) = *hole-pos C*
by (*induct C*) *auto*

lemma *hole-pos-in-ctxt-apply* [*simp*]: *hole-pos C* ∈ *poss C*⟨*u*⟩
by (*induct C*) *auto*

2.3 Properties of *Term-Context.ground* and *ground-ctxt*

lemma *ground-vars-term-empty* [*simp*]:
ground t ⟹ *vars-term t* = {}
by (*induct t*) *auto*

lemma *ground-map-term* [*simp*]:
ground (*map-term f h t*) = *ground t*
by (*induct t*) *auto*

lemma *ground-ctxt-apply* [*simp*]:
ground C⟨*t*⟩ ⟷ *ground-ctxt C* ∧ *ground t*
by (*induct C*) *auto*

lemma *ground-ctxt-comp* [*intro*]:
ground-ctxt C ⟹ *ground-ctxt D* ⟹ *ground-ctxt* (*C* ∘$_c$ *D*)
by (*induct C*) *auto*

lemma *ctxt-comp-n-pres-ground* [*intro*]:
ground-ctxt C ⟹ *ground-ctxt* (*C*^*n*)
by (*induct n arbitrary*: *C*) *auto*

lemma *subterm-eq-pres-ground*:
assumes *ground s* **and** *s* ⊵ *t*
shows *ground t* **using** *assms*(*2*,*1*)
by (*induct*) *auto*

lemma *ground-substD*:
ground (*l* · *σ*) ⟹ *x* ∈ *vars-term l* ⟹ *ground* (*σ x*)
by (*induct l*) *auto*

lemma *ground-substI*:

$(\bigwedge x.\ x \in \mathit{vars\text{-}term}\ s \Longrightarrow \mathit{ground}\ (\sigma\ x)) \Longrightarrow \mathit{ground}\ (s \cdot \sigma)$
by (*induct s*) *auto*

2.4 Properties on signature induced by a term *Term.term*/context *ctxt*

lemma *funas-ctxt-apply* [*simp*]:
$\mathit{funas\text{-}term}\ C\langle t\rangle = \mathit{funas\text{-}ctxt}\ C \cup \mathit{funas\text{-}term}\ t$
by (*induct C*) *auto*

lemma *funas-term-map* [*simp*]:
$\mathit{funas\text{-}term}\ (\mathit{map\text{-}term}\ f\ h\ t) = (\lambda\ (g,\ n).\ (f\ g,\ n))\ `\ \mathit{funas\text{-}term}\ t$
by (*induct t*) *auto*

lemma *funas-term-subst*:
$\mathit{funas\text{-}term}\ (l \cdot \sigma) = \mathit{funas\text{-}term}\ l \cup (\bigcup\ (\mathit{funas\text{-}term}\ `\ \sigma\ `\ \mathit{vars\text{-}term}\ l))$
by (*induct l*) *auto*

lemma *funas-ctxt-comp* [*simp*]:
$\mathit{funas\text{-}ctxt}\ (C \circ_c D) = \mathit{funas\text{-}ctxt}\ C \cup \mathit{funas\text{-}ctxt}\ D$
by (*induct C*) *auto*

lemma *ctxt-comp-n-funas* [*simp*]:
$(f,\ v) \in \mathit{funas\text{-}ctxt}\ (C\hat{}n) \Longrightarrow (f,\ v) \in \mathit{funas\text{-}ctxt}\ C$
by (*induct n arbitrary*: *C*) *auto*

lemma *ctxt-comp-n-pres-funas* [*intro*]:
$\mathit{funas\text{-}ctxt}\ C \subseteq \mathcal{F} \Longrightarrow \mathit{funas\text{-}ctxt}\ (C\hat{}n) \subseteq \mathcal{F}$
by (*induct n arbitrary*: *C*) *auto*

2.5 Properties on subterm at given position (|-)

lemma *subt-at-Cons-comp*:
$i\ \#\ p \in \mathit{poss}\ s \Longrightarrow (s\ |\text{-}\ [i])\ |\text{-}\ p = s\ |\text{-}\ (i\ \#\ p)$
by (*cases s*) *auto*

lemma *ctxt-at-pos-subt-at-pos*:
$p \in \mathit{poss}\ t \Longrightarrow (\mathit{ctxt\text{-}at\text{-}pos}\ t\ p)\langle u\rangle\ |\text{-}\ p = u$
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
then show *?case* **using** *id-take-nth-drop*
by (*cases t*) (*auto simp*: *nth-append*)
qed *auto*

lemma *ctxt-at-pos-subt-at-id*:
$p \in \mathit{poss}\ t \Longrightarrow (\mathit{ctxt\text{-}at\text{-}pos}\ t\ p)\langle t\ |\text{-}\ p\rangle = t$
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
then show *?case* **using** *id-take-nth-drop*

by (*cases t*) *force+*
qed *auto*

lemma *subst-at-ctxt-at-eq-termD*:
assumes $s = t$ $p \in$ *poss t*
shows s |- $p = t$ |- $p \wedge$ *ctxt-at-pos s p* = *ctxt-at-pos t p* **using** *assms*
by *auto*

lemma *subst-at-ctxt-at-eq-termI*:
assumes $p \in$ *poss s* $p \in$ *poss t*
and s |-$p = t$ |- p
and *ctxt-at-pos s p* = *ctxt-at-pos t p*
shows $s = t$ **using** *assms*
by (*metis ctxt-at-pos-subt-at-id*)

lemma *subt-at-subterm-eq* [*intro*!]:
$p \in$ *poss t* $\Longrightarrow$ $t \trianglerighteq t$ |- p
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
from *Cons*(*1*)[*of args t* ! *i*] *Cons*(*2*) **show** *?case*
by (*cases t*) *force+*
qed *auto*

lemma *subt-at-subterm* [*intro*!]:
$p \in$ *poss t* $\Longrightarrow$ $p \neq []$ $\Longrightarrow$ $t \triangleright t$ |- p
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
from *Cons*(*1*)[*of args t* ! *i*] *Cons*(*2*) **show** *?case*
by (*cases t*) *force+*
qed *auto*

lemma *ctxt-at-pos-hole-pos* [*simp*]: *ctxt-at-pos* $C\langle s\rangle$ (*hole-pos C*) = *C*
by (*induct C*) *auto*

2.6 Properties on replace terms at a given position *replace-term-at*

lemma *replace-term-at-not-poss* [*simp*]:
$p \notin$ *poss s* $\Longrightarrow$ $s[p \leftarrow t] = s$
proof (*induct s arbitrary*: *p*)
case (*Var x*) **then show** *?case* **by** (*cases p*) *auto*
next
case (*Fun f ts*) **show** *?case* **using** *Fun*(*1*)[*OF nth-mem*] *Fun*(*2*)
by (*cases p*) (*auto simp*: *min-def intro*!: *nth-equalityI*)
qed

lemma *replace-term-at-replace-at-conv*:
$p \in$ *poss s* $\Longrightarrow$ (*ctxt-at-pos s p*)$\langle t\rangle$ = $s[p \leftarrow t]$
by (*induct s arbitrary*: *p*) (*auto simp*: *upd-conv-take-nth-drop*)

lemma *parallel-replace-term-commute* [*ac-simps*]:
  $p \perp q \Longrightarrow s[p \leftarrow t][q \leftarrow u] = s[q \leftarrow u][p \leftarrow t]$
proof (*induct s arbitrary*: *p q*)
  case (*Var x*) **then show** *?case*
    by (*cases p*; *cases q*) *auto*
next
  case (*Fun f ts*)
  from *Fun*(*2*) **have** $p \neq []$ $q \neq []$ **by** *auto*
  then obtain *i j ps qs* **where** [*simp*]: $p = i \# ps$ $q = j \# qs$
    by (*cases p*; *cases q*) *auto*
  have $i \neq j \Longrightarrow (Fun\ f\ ts)[p \leftarrow t][q \leftarrow u] = (Fun\ f\ ts)[q \leftarrow u][p \leftarrow t]$
    by (*auto simp*: *list-update-swap*)
  then show *?case* **using** *Fun*(*1*)[*OF nth-mem*, *of j ps qs*] *Fun*(*2*)
    by (*cases i* = *j*) (*auto simp*: *par-Cons-iff*)
qed

lemma *replace-term-at-above* [*simp*]:
  $p \leq_p q \Longrightarrow s[q \leftarrow t][p \leftarrow u] = s[p \leftarrow u]$
proof (*induct p arbitrary*: *s q*)
  case (*Cons i p*)
  show *?case* **using** *Cons*(*1*)[*of tl q args s* ! *i*] *Cons*(*2*)
    by (*cases q*; *cases s*) *auto*
qed *auto*

lemma *replace-term-at-below* [*simp*]:
  $p <_p q \Longrightarrow s[p \leftarrow t][q \leftarrow u] = s[p \leftarrow t[q -_p p \leftarrow u]]$
proof (*induct p arbitrary*: *s q*)
  case (*Cons i p*)
  show *?case* **using** *Cons*(*1*)[*of tl q args s* ! *i*] *Cons*(*2*)
    by (*cases q*; *cases s*) *auto*
qed *auto*

lemma *replace-at-hole-pos* [*simp*]: $C\langle s\rangle[hole\text{-}pos\ C \leftarrow t] = C\langle t\rangle$
  by (*induct C*) *auto*

2.7 Properties on *adapt-vars* and *adapt-vars-ctxt*

lemma *adapt-vars2*:
  adapt-vars (*adapt-vars t*) = *adapt-vars t*
  by (*induct t*) (*auto simp add*: *adapt-vars-def*)

lemma *adapt-vars-simps*[*code*, *simp*]: *adapt-vars* (*Fun f ts*) = *Fun f* (*map adapt-vars ts*)
  by (*induct ts*, *auto simp*: *adapt-vars-def*)

lemma *adapt-vars-reverse*: *ground t* $\Longrightarrow$ *adapt-vars t'* = *t* $\Longrightarrow$ *adapt-vars t* = *t'*
  unfolding *adapt-vars-def*
proof (*induct t arbitrary*: *t'*)

case (*Fun f ts*)
then show *?case* **by** (*cases t′*) (*auto simp add*: *map-idI*)
qed *auto*

lemma *ground-adapt-vars* [*simp*]: *ground* (*adapt-vars t*) = *ground t*
by (*simp add*: *adapt-vars-def*)
lemma *funas-term-adapt-vars*[*simp*]: *funas-term* (*adapt-vars t*) = *funas-term t* **by** (*simp add*: *adapt-vars-def*)

lemma *adapt-vars-adapt-vars*[*simp*]: **fixes** *t* :: (*′f*,*′v*)*term*
assumes *g*: *ground t*
shows *adapt-vars* (*adapt-vars t* :: (*′f*,*′w*)*term*) = *t*
proof −
let *?t′* = *adapt-vars t* :: (*′f*,*′w*)*term*
have *gt′*: *ground ?t′* **using** *g* **by** *auto*
from *adapt-vars-reverse*[*OF gt′*, *of t*] **show** *?thesis* **by** *blast*
qed

lemma *adapt-vars-inj*:
assumes *adapt-vars x* = *adapt-vars y ground x ground y*
shows *x* = *y*
using *adapt-vars-adapt-vars assms* **by** *metis*

lemma *adapt-vars-ctxt-simps*[*simp*, *code*]:
adapt-vars-ctxt (*More f bef C aft*) = *More f* (*map adapt-vars bef*) (*adapt-vars-ctxt C*) (*map adapt-vars aft*)
adapt-vars-ctxt Hole = *Hole* **unfolding** *adapt-vars-ctxt-def adapt-vars-def* **by** *auto*

lemma *adapt-vars-ctxt*[*simp*]: *adapt-vars* (*C* ⟨ *t* ⟩) = (*adapt-vars-ctxt C*) ⟨ *adapt-vars t* ⟩
by (*induct C*, *auto*)

lemma *adapt-vars-subst*[*simp*]: *adapt-vars* (*l* · *σ*) = *l* · (*λ x*. *adapt-vars* (*σ x*))
unfolding *adapt-vars-def*
by (*induct l*) *auto*

lemma *adapt-vars-gr-map-vars* [*simp*]:
ground t ⟹ *map-vars-term f t* = *adapt-vars t*
by (*induct t*) *auto*

lemma *adapt-vars-gr-ctxt-of-map-vars* [*simp*]:
ground-ctxt C ⟹ *map-vars-ctxt f C* = *adapt-vars-ctxt C*
by (*induct C*) *auto*

2.7.1 Equality on ground terms/contexts by positions and symbols

lemma *fun-at-def'*:
fun-at t p = (*if p* ∈ *poss t then*
(*case t* |- *p of Var x* ⇒ *Some* (*Inr x*) | *Fun f ts* ⇒ *Some* (*Inl f*)) *else None*)
by (*induct t p rule*: *fun-at.induct*) *auto*

lemma *fun-at-None-nposs-iff*:
fun-at t p = *None* ⟷ *p* ∉ *poss t*
by (*auto simp*: *fun-at-def'*) (*meson term.case-eq-if*)

lemma *eq-term-by-poss-fun-at*:
assumes *poss s* = *poss t* ⋀*p*. *p* ∈ *poss s* ⟹ *fun-at s p* = *fun-at t p*
shows *s* = *t*
using *assms*
proof (*induct s arbitrary*: *t*)
case (*Var x*) **then show** *?case*
by (*cases t*) *simp-all*
next
case (*Fun f ss*) **note** *Fun'* = *this*
show *?case*
proof (*cases t*)
case (*Var x*) **show** *?thesis* **using** *Fun'*(*3*)[*of* []] **by** (*simp add*: *Var*)
next
case (*Fun g ts*)
have ∗: *length ss* = *length ts*
using *Fun'*(*3*) *arg-cong*[*OF Fun'*(*2*), *of* λ*P*. *card* {*i* |*i p*. *i* # *p* ∈ *P*}]
by (*auto simp*: *Fun exI*[*of* λ*x*. *x* ∈ *poss* -, *OF Nil-in-poss*])
then have *i* < *length ss* ⟹ *poss* (*ss* ! *i*) = *poss* (*ts* ! *i*) **for** *i*
using *arg-cong*[*OF Fun'*(*2*), *of* λ*P*. {*p*. *i* # *p* ∈ *P*}] **by** (*auto simp*: *Fun*)
then show *?thesis* **using** ∗ *Fun'*(*2*) *Fun'*(*3*)[*of* []] *Fun'*(*3*)[*of* - # - :: *pos*]
by (*auto simp*: *Fun intro*!: *nth-equalityI Fun'*(*1*)[*OF nth-mem*, *of n ts* ! *n* **for** *n*])
qed
qed

lemma *eq-ctxt-at-pos-by-poss*:
assumes *p* ∈ *poss s p* ∈ *poss t*
and ⋀ *q*. ¬ (*p* ≤*p q*) ⟹ *q* ∈ *poss s* ⟷ *q* ∈ *poss t*
and (⋀ *q*. *q* ∈ *poss s* ⟹ ¬ (*p* ≤*p q*) ⟹ *fun-at s q* = *fun-at t q*)
shows *ctxt-at-pos s p* = *ctxt-at-pos t p* **using** *assms*
proof (*induct p arbitrary*: *s t*)
case (*Cons i p*)
from *Cons*(*2*, *3*) *Cons*(*4*, *5*)[*of* []] **obtain** *f ss ts* **where** [*simp*]: *s* = *Fun f ss t* = *Fun f ts*
by (*cases s*; *cases t*) *auto*
have *flt*: *j* < *i* ⟹ *j* # *q* ∈ *poss s* ⟹ *fun-at s* (*j* # *q*) = *fun-at t* (*j* # *q*) **for** *j q*
by (*intro Cons*(*5*)) *auto*
have *fgt*: *i* < *j* ⟹ *j* # *q* ∈ *poss s* ⟹ *fun-at s* (*j* # *q*) = *fun-at t* (*j* # *q*) **for** *j*

q
by (*intro Cons*(*5*)) *auto*
have *lt*: $j < i \Longrightarrow j \# q \in poss\ s \longleftrightarrow j \# q \in poss\ t$ **for** *j q* **by** (*intro Cons*(*4*)) *auto*
have *gt*: $i < j \Longrightarrow j \# q \in poss\ s \longleftrightarrow j \# q \in poss\ t$ **for** *j q* **by** (*intro Cons*(*4*)) *auto*
from *this*[*of* - []] **have** $i < j \Longrightarrow j < length\ ss \longleftrightarrow j < length\ ts$ **for** *j* **by** *auto*
from *this Cons*(*2*, *3*) **have** *l*: $length\ ss = length\ ts$ **by** *auto* (*meson nat-neq-iff*)
have $ctxt\text{-}at\text{-}pos\ (ss\ !\ i)\ p = ctxt\text{-}at\text{-}pos\ (ts\ !\ i)\ p$ **using** *Cons*(*2*, *3*) *Cons*(*4*−)[*of i # q* **for** *q*]
by (*intro Cons*(*1*)[*of ss* ! *i ts* ! *i*]) *auto*
moreover have $take\ i\ ss = take\ i\ ts$ **using** *l lt Cons*(*2*, *3*) *flt*
by (*intro nth-equalityI*) (*auto intro*!: *eq-term-by-poss-fun-at*)
moreover have $drop\ (Suc\ i)\ ss = drop\ (Suc\ i)\ ts$ **using** *l Cons*(*2*, *3*) *fgt gt*[*of Suc i* + *j* **for** *j*]
by (*intro nth-equalityI*) (*auto simp*: *nth-map intro*!: *eq-term-by-poss-fun-at*, *fastforce*+)
ultimately show *?case* **by** *auto*
qed *auto*

2.8 Misc

lemma *fun-at-hole-pos-ctxt-apply* [*simp*]:
$fun\text{-}at\ C\langle t\rangle\ (hole\text{-}pos\ C) = fun\text{-}at\ t\ []$
by (*induct C*) *auto*

lemma *vars-term-ctxt-apply* [*simp*]:
$vars\text{-}term\ C\langle t\rangle = vars\text{-}ctxt\ C \cup vars\text{-}term\ t$
by (*induct C arbitrary*: *t*) *auto*

lemma *map-vars-term-ctxt-apply*:
$map\text{-}vars\text{-}term\ f\ C\langle t\rangle = (map\text{-}vars\text{-}ctxt\ f\ C)\langle map\text{-}vars\text{-}term\ f\ t\rangle$
by (*induct C*) *auto*

lemma *map-term-replace-at-dist*:
$p \in poss\ s \Longrightarrow (map\text{-}term\ f\ g\ s)[p \leftarrow (map\text{-}term\ f\ g\ t)] = map\text{-}term\ f\ g\ (s[p \leftarrow t])$
proof (*induct p arbitrary*: *s*)
case (*Cons i p*) **then show** *?case*
by (*cases s*) (*auto simp*: *nth-list-update intro*!: *nth-equalityI*)
qed *auto*

end
theory *Basic-Utils*
imports *Term-Context*
begin

primrec *is-Inl* **where**
$is\text{-}Inl\ (Inl\ q) \longleftrightarrow True$

| *is-Inl* (*Inr q*) ⟷ *False*

primrec *is-Inr* **where**
  is-Inr (*Inr q*) ⟷ *True*
| *is-Inr* (*Inl q*) ⟷ *False*

fun *remove-sum* **where**
  remove-sum (*Inl q*) = *q*
| *remove-sum* (*Inr q*) = *q*

List operations

definition *filter-rev-nth* **where**
  filter-rev-nth P xs i = *length* (*filter P* (*take* (*Suc i*) *xs*)) − *1*

lemma *filter-rev-nth-butlast*:
  ¬ *P* (*last xs*) ⟹ *filter-rev-nth P xs i* = *filter-rev-nth P* (*butlast xs*) *i*
  unfolding *filter-rev-nth-def*
  by (*induct xs arbitrary*: *i rule*: *rev-induct*) (*auto simp add*: *take-Cons′*)

lemma *filter-rev-nth-idx*:
  assumes *i* < *length xs P* (*xs* ! *i*) *ys* = *filter P xs*
  shows *xs* ! *i* = *ys* ! (*filter-rev-nth P xs i*) ∧ *filter-rev-nth P xs i* < *length ys*
  using *assms* **unfolding** *filter-rev-nth-def*
proof (*induct xs arbitrary*: *ys i*)
  case (*Cons x xs*) **show** *?case*
  proof (*cases P x*)
    case *True*
    then obtain *ys′* **where** ∗:*ys* = *x* # *ys′* **using** *Cons*(*4*) **by** *auto*
    show *?thesis* **using** *True Cons*(*1*)[*of i* − *1 ys′*] *Cons*(*2*−)
      unfolding ∗
      by (*cases i*) (*auto simp*: *nth-Cons′ take-Suc-conv-app-nth*)
  next
    case *False*
    then show *?thesis* **using** *Cons*(*1*)[*of i* − *1 ys*] *Cons*(*2*−)
      by (*auto simp*: *nth-Cons′*)
  qed
qed *auto*

primrec *add-elem-list-lists* :: *′a* ⇒ *′a list* ⇒ *′a list list* **where**
  add-elem-list-lists x [] = [[*x*]]
| *add-elem-list-lists x* (*y* # *ys*) = (*x* # *y* # *ys*) # (*map* ((#) *y*) (*add-elem-list-lists x ys*))

lemma *length-add-elem-list-lists*:
  ys ∈ *set* (*add-elem-list-lists x xs*) ⟹ *length ys* = *Suc* (*length xs*)
  by (*induct xs arbitrary*: *ys*) *auto*

lemma *add-elem-list-listsE*:
assumes *ys* ∈ *set* (*add-elem-list-lists x xs*)
shows ∃ *n* ≤ *length xs*. *ys* = *take n xs* @ *x* # *drop n xs* **using** *assms*
proof(*induct xs arbitrary*: *ys*)
case (*Cons a xs*)
then show *?case*
by *auto fastforce*
qed *auto*

lemma *add-elem-list-listsI*:
assumes *n* ≤ *length xs ys* = *take n xs* @ *x* # *drop n xs*
shows *ys* ∈ *set* (*add-elem-list-lists x xs*) **using** *assms*
proof (*induct xs arbitrary*: *ys n*)
case (*Cons a xs*)
then show *?case*
by (*cases n*) (*auto simp*: *image-iff*)
qed *auto*

lemma *add-elem-list-lists-def'*:
set (*add-elem-list-lists x xs*) = {*ys* | *ys n*. *n* ≤ *length xs* ∧ *ys* = *take n xs* @ *x* # *drop n xs*}
using *add-elem-list-listsI add-elem-list-listsE*
by *fastforce*

fun *list-of-permutation-element-n* :: *'a* ⇒ *nat* ⇒ *'a list* ⇒ *'a list list* **where**
list-of-permutation-element-n x 0 L = [[]]
| *list-of-permutation-element-n x* (*Suc n*) *L* = *concat* (*map* (*add-elem-list-lists x*) (*List.n-lists n L*))

lemma *list-of-permutation-element-n-conv*:
assumes *n* ≠ *0*
shows *set* (*list-of-permutation-element-n x n L*) =
{*xs* | *xs i*. *i* < *length xs* ∧ (∀ *j* < *length xs*. *j* ≠ *i* ⟶ *xs* ! *j* ∈ *set L*) ∧ *length xs* = *n* ∧ *xs* ! *i* = *x*} (**is** *?Ls* = *?Rs*)
proof (*intro equalityI*)
from *assms* **obtain** *j* **where** [*simp*]: *n* = *Suc j* **using** *assms* **by** (*cases n*) *auto*
{fix *ys* **assume** *ys* ∈ *?Ls*
then obtain *xs i* **where** *wit*: *xs* ∈ *set* (*List.n-lists j L*) *i* ≤ *length xs*
ys = *take i xs* @ *x* # *drop i xs*
by (*auto dest*: *add-elem-list-listsE*)
then have *i* < *length ys length ys* = *Suc* (*length xs*) *ys* ! *i* = *x*
by (*auto simp*: *nth-append*)
moreover have ∀ *j* < *length ys*. *j* ≠ *i* ⟶ *ys* ! *j* ∈ *set L* **using** *wit*(*1*, *2*)
by (*auto simp*: *wit*(*3*) *min-def nth-append set-n-lists*)
ultimately have *ys* ∈ *?Rs* **using** *wit*(*1*) **unfolding** *set-n-lists*
by *auto***}**
then show *?Ls* ⊆ *?Rs* **by** *blast*
next

{fix *xs* **assume** *xs* ∈ *?Rs*
then obtain *i* **where** *wit*: *i* < *length xs* ∀ *j* < *length xs*. *j* ≠ *i* ⟶ *xs* ! *j* ∈ *set L*
length xs = *n xs* ! *i* = *x*
by *blast*
then have ∗: *xs* ∈ *set* (*add-elem-list-lists* (*xs* ! *i*) (*take i xs* @ *drop* (*Suc i*) *xs*))
unfolding *add-elem-list-lists-def′*
by (*auto simp*: *min-def intro*!: *nth-equalityI*)
(*metis Cons-nth-drop-Suc Suc-pred append-Nil append-take-drop-id assms diff-le-self diff-self-eq-0 drop-take less-Suc-eq-le nat-less-le take0*)
have [*simp*]: *x* ∈ *set* (*take i xs*) ⟹ *x* ∈ *set L*
x ∈ *set* (*drop* (*Suc i*) *xs*) ⟹ *x* ∈ *set L* **for** *x* **using** *wit*(*2*)
by (*auto simp*: *set-conv-nth*)
have *xs* ∈ *?Ls* **using** *wit*
by (*cases length xs*)
(*auto simp*: *set-n-lists nth-append* ∗ *min-def*
intro!: *exI*[*of - take i xs* @ *drop* (*Suc i*) *xs*])**}**
then show *?Rs* ⊆ *?Ls* **by** *blast*
qed

lemma *list-of-permutation-element-n-iff*:
set (*list-of-permutation-element-n x n L*) =
(*if n* = *0 then* {[]} *else* {*xs* | *xs i*. *i* < *length xs* ∧ (∀ *j* < *length xs*. *j* ≠ *i* ⟶ *xs* ! *j* ∈ *set L*) ∧ *length xs* = *n* ∧ *xs* ! *i* = *x*})
proof (*cases n*)
case (*Suc nat*)
then have [*simp*]: *Suc nat* ≠ *0* **by** *auto*
then show *?thesis*
by (*auto simp*: *list-of-permutation-element-n-conv*)
qed *auto*

lemma *list-of-permutation-element-n-conv′*:
assumes *x* ∈ *set L 0* < *n*
shows *set* (*list-of-permutation-element-n x n L*) =
{*xs*. *set xs* ⊆ *insert x* (*set L*) ∧ *length xs* = *n* ∧ *x* ∈ *set xs*}
proof −
from *assms*(*2*) **have** ∗: *n* ≠ *0* **by** *simp*
show *?thesis* **using** *assms*
unfolding *list-of-permutation-element-n-conv*[*OF* ∗]
by (*auto simp*: *in-set-conv-nth*)
(*metis in-set-conv-nth insert-absorb subsetD*)+
qed

Misc

lemma *in-set-idx*:
x ∈ *set xs* ⟹ ∃ *i* < *length xs*. *xs* ! *i* = *x*
by (*induct xs*) *force*+

lemma *set-list-subset-eq-nth-conv*:

set xs $\subseteq$ *A* $\longleftrightarrow$ ($\forall$ *i* < *length xs*. *xs* ! *i* $\in$ *A*)
by (*metis in-set-conv-nth subset-code*(*1*))

lemma *map-eq-nth-conv*:
map f xs = *map g ys* $\longleftrightarrow$ *length xs* = *length ys* $\wedge$ ($\forall$ *i* < *length ys*. *f* (*xs* ! *i*) = *g* (*ys* ! *i*))
using *map-eq-imp-length-eq*[*of f xs g ys*]
by (*auto intro*: *nth-equalityI*) (*metis nth-map*)

lemma *nth-append-Cons*: (*xs* @ *y* # *zs*) ! *i* =
(*if i* < *length xs then xs* ! *i else if i* = *length xs then y else zs* ! (*i* − *Suc* (*length xs*)))
by (*cases i length xs rule*: *linorder-cases*, *auto simp*: *nth-append*)

lemma *map-prod-times*:
f ‘ *A* × *g* ‘ *B* = *map-prod f g* ‘ (*A* × *B*)
by *auto*

lemma *trancl-full-on*: (*X* × *X*)$^+$ = *X* × *X*
using *trancl-unfold-left*[*of X* × *X*] *trancl-unfold-right*[*of X* × *X*] **by** *auto*

lemma *trancl-map*:
assumes *simu*: $\bigwedge$*x y*. (*x*, *y*) $\in$ *r* $\Longrightarrow$ (*f x*, *f y*) $\in$ *s*
and *steps*: (*x*, *y*) $\in$ *r*$^+$
shows (*f x*, *f y*) $\in$ *s*$^+$ **using** *steps*
proof (*induct*)
case (*step y z*) **show** *?case* **using** *step*(*3*) *simu*[*OF step*(*2*)]
by *auto*
qed (*auto simp*: *simu*)

lemma *trancl-map-prod-mono*:
map-both f ‘ *R*$^+$ $\subseteq$ (*map-both f* ‘ *R*)$^+$
proof −
have (*f x*, *f y*) $\in$ (*map-both f* ‘ *R*)$^+$ **if** (*x*, *y*) $\in$ *R*$^+$ **for** *x y* **using** *that*
by (*induct*) (*auto intro*: *trancl-into-trancl*)
then show *?thesis* **by** *auto*
qed

lemma *trancl-map-both-Restr*:
assumes *inj-on f X*
shows (*map-both f* ‘ *Restr R X*)$^+$ = *map-both f* ‘ (*Restr R X*)$^+$
proof −
have [*simp*]:
map-prod (*inv-into X f* ∘ *f*) (*inv-into X f* ∘ *f*) ‘ *Restr R X* = *Restr R X*
using *inv-into-f-f*[*OF assms*]
by (*intro equalityI subrelI*)
(*force simp*: *comp-def map-prod-def image-def split*: *prod.splits*)+
have [*simp*]:
map-prod (*f* ∘ *inv-into X f*) (*f* ∘ *inv-into X f*) ‘ (*map-both f* ‘ *Restr R X*)$^+$ =

(*map-both f ‘ Restr R X*)$^+$
using *f-inv-into-f*[*of - f X*] *subsetD*[*OF trancl-mono-set*[*OF image-mono*[*of Restr R X X* × *X map-both f*]]]
by (*intro equalityI subrelI*) (*auto simp*: *map-prod-surj-on trancl-full-on comp-def rev-image-eqI*)
show *?thesis* **using** *assms trancl-map-prod-mono*[*of f Restr R X*] *image-mono*[*OF trancl-map-prod-mono*[*of inv-into X f map-both f ‘ Restr R X*], *of map-both f*]
by (*intro equalityI*) (*simp-all add*: *image-comp map-prod.comp*)
qed

lemma *inj-on-trancl-map-both*:
assumes *inj-on f* (*fst ‘ R* ∪ *snd ‘ R*)
shows (*map-both f ‘ R*)$^+$ = *map-both f ‘ R*$^+$
proof −
have [*simp*]: *Restr R* (*fst ‘ R* ∪ *snd ‘ R*) = *R*
by (*force simp*: *image-def*)
then show *?thesis* **using** *assms*
using *trancl-map-both-Restr*[*of f fst ‘ R* ∪ *snd ‘ R R*]
by *simp*
qed

lemma *kleene-induct*:
A ⊆ *X* ⟹ *B O X* ⊆ *X* ⟹ *X O C* ⊆ *X* ⟹ *B** *O A O C** ⊆ *X*
using *relcomp-mono*[*OF compat-tr-compat*[*of B X*] *subset-refl*, *of C**] *compat-tr-compat*[*of C*$^{-1}$ *X*$^{-1}$]
relcomp-mono[*OF relcomp-mono*, *OF subset-refl - subset-refl*, *of A X B** *C**]
unfolding *rtrancl-converse converse-relcomp*[*symmetric*] *converse-mono* **by** *blast*

lemma *kleene-trancl-induct*:
A ⊆ *X* ⟹ *B O X* ⊆ *X* ⟹ *X O C* ⊆ *X* ⟹ *B*$^+$ *O A O C*$^+$ ⊆ *X*
using *kleene-induct*[*of A X B C*]
by (*auto simp*: *rtrancl-eq-or-trancl*)
(*meson relcomp.relcompI subsetD trancl-into-rtrancl*)

lemma *rtrancl-Un2-separatorE*:
B O A = {} ⟹ (*A* ∪ *B*)* = *A** ∪ *A** *O B**
by (*metis R-O-Id empty-subsetI relcomp-distrib rtrancl-U-push rtrancl-reflcl-absorb sup-commute*)

lemma *trancl-Un2-separatorE*:
assumes *B O A* = {}
shows (*A* ∪ *B*)$^+$ = *A*$^+$ ∪ *A*$^+$ *O B*$^+$ ∪ *B*$^+$ (**is** *?Ls* = *?Rs*)
proof −
{fix *x y* **assume** (*x*, *y*) ∈ *?Ls*
then have (*x*, *y*) ∈ *?Rs* **using** *assms*
proof (*induct*)
case (*step y z*)

```
    then show ?case
      by (auto simp add: trancl-into-trancl relcomp-unfold dest: tranclD2)
   qed auto}
  then show ?thesis
   by (auto simp add: trancl-mono)
     (meson sup-ge1 sup-ge2 trancl-mono trancl-trans)
qed
```

Sum types where both components have the same type (to create copies)

```
lemma is-InrE:
  assumes is-Inr q
  obtains p where q = Inr p
  using assms by (cases q) auto

lemma is-InlE:
  assumes is-Inl q
  obtains p where q = Inl p
  using assms by (cases q) auto

lemma not-is-Inr-is-Inl [simp]:
  ¬ is-Inl t ⟷ is-Inr t
  ¬ is-Inr t ⟷ is-Inl t
  by (cases t, auto)+

lemma [simp]: remove-sum ∘ Inl = id by auto

abbreviation CInl :: 'q ⇒ 'q + 'q where CInl ≡ Inl
abbreviation CInr :: 'q ⇒ 'q + 'q where CInr ≡ Inr

lemma inj-CInl: inj CInl inj CInr using inj-Inl inj-Inr by blast+

lemma map-prod-simp': map-prod f g G = (f (fst G), g (snd G))
  by (auto simp add: map-prod-def split!: prod.splits)

end
```

2.9 Ground constructions

```
theory Ground-Terms
  imports Basic-Utils
begin
```

2.9.1 Ground terms

This type serves two purposes. First of all, the encoding definitions and proofs are not littered by cases for variables. Secondly, we can consider tree domains (usually sets of positions), which become a special case of ground terms. This enables the construction of a term from a tree domain and a function from positions to symbols.

datatype *'f gterm* =
  GFun (*groot-sym*: *'f*) (*gargs*: *'f gterm list*)

lemma *gterm-idx-induct*[*case-names GFun*]:
  assumes $\bigwedge f\ ts.\ (\bigwedge i.\ i < length\ ts \Longrightarrow P\ (ts\ !\ i)) \Longrightarrow P\ (GFun\ f\ ts)$
  shows *P t* **using** *assms*
  by (*induct t*) *auto*

fun *term-of-gterm* **where**
  term-of-gterm (*GFun f ts*) = *Fun f* (*map term-of-gterm ts*)

fun *gterm-of-term* **where**
  gterm-of-term (*Fun f ts*) = *GFun f* (*map gterm-of-term ts*)

fun *groot* **where**
  groot (*GFun f ts*) = (*f*, *length ts*)

lemma *groot-sym-groot-conv*:
  groot-sym t = *fst* (*groot t*)
  by (*cases t*) *auto*

lemma *groot-sym-gterm-of-term*:
  ground t $\Longrightarrow$ *groot-sym* (*gterm-of-term t*) = *fst* (*the* (*root t*))
  by (*cases t*) *auto*

lemma *length-args-length-gargs* [*simp*]:
  length (*args* (*term-of-gterm t*)) = *length* (*gargs t*)
  by (*cases t*) *auto*

lemma *ground-term-of-gterm* [*simp*]:
  ground (*term-of-gterm s*)
  by (*induct s*) *auto*

lemma *ground-term-of-gterm'* [*simp*]:
  term-of-gterm s = *Fun f ss* $\Longrightarrow$ *ground* (*Fun f ss*)
  by (*induct s*) *auto*

lemma *term-of-gterm-inv* [*simp*]:
  gterm-of-term (*term-of-gterm t*) = *t*
  by (*induct t*) (*auto intro*!: *nth-equalityI*)

lemma *inj-term-of-gterm*:
  inj-on term-of-gterm X
  by (*metis inj-on-def term-of-gterm-inv*)

lemma *gterm-of-term-inv* [*simp*]:
  ground t $\Longrightarrow$ *term-of-gterm* (*gterm-of-term t*) = *t*
  by (*induct t*) (*auto 0 0 intro*!: *nth-equalityI*)

lemma *ground-term-to-gtermD*:
ground t $\Longrightarrow \exists\, t'.\ t =$ *term-of-gterm t'*
by (*metis gterm-of-term-inv*)

lemma *map-term-of-gterm* [*simp*]:
map-term f g (*term-of-gterm t*) = *term-of-gterm* (*map-gterm f t*)
by (*induct t*) *auto*

lemma *map-gterm-of-term* [*simp*]:
ground t $\Longrightarrow$ *gterm-of-term* (*map-term f g t*) = *map-gterm f* (*gterm-of-term t*)
by (*induct t*) *auto*

lemma *gterm-set-gterm-funs-terms*:
set-gterm t = *funs-term* (*term-of-gterm t*)
by (*induct t*) *auto*

lemma *term-set-gterm-funs-terms*:
assumes *ground t*
shows *set-gterm* (*gterm-of-term t*) = *funs-term t*
using *assms* **by** (*induct t*) *auto*

lemma *vars-term-of-gterm* [*simp*]:
vars-term (*term-of-gterm t*) = {}
by (*induct t*) *auto*

lemma *vars-term-of-gterm-subseteq* [*simp*]:
vars-term (*term-of-gterm t*) $\subseteq$ *Q* $\longleftrightarrow$ *True*
by *auto*

context
notes *conj-cong* [*fundef-cong*]
begin
fun *gposs* :: *'f gterm* $\Rightarrow$ *pos set* **where**
gposs (*GFun f ss*) = {[]} $\cup$ {*i* # *p* | *i p*. *i* < *length ss* $\wedge$ *p* $\in$ *gposs* (*ss* ! *i*)}
end

lemma *gposs-Nil* [*simp*]: [] $\in$ *gposs s*
by (*cases s*) *auto*

lemma *gposs-map-gterm* [*simp*]:
gposs (*map-gterm f s*) = *gposs s*
by (*induct s*) *auto*

lemma *poss-gposs-conv*:
poss (*term-of-gterm t*) = *gposs t*
by (*induct t*) *auto*

lemma *poss-gposs-mem-conv*:
p $\in$ *poss* (*term-of-gterm t*) $\longleftrightarrow$ *p* $\in$ *gposs t*

using *poss-gposs-conv* **by** *auto*

lemma *gposs-to-poss*:
p ∈ *gposs t* ⟹ *p* ∈ *poss* (*term-of-gterm t*)
by (*simp add*: *poss-gposs-mem-conv*)

fun *gfun-at* :: *'f gterm* ⇒ *pos* ⇒ *'f option* **where**
gfun-at (*GFun f ts*) [] = *Some f*
| *gfun-at* (*GFun f ts*) (*i* # *p*) = (*if i* < *length ts then gfun-at* (*ts* ! *i*) *p else None*)

abbreviation *exInl* ≡ *case-sum* (λ *x*. *x*) (λ -.*undefined*)

lemma *gfun-at-gterm-of-term* [*simp*]:
ground s ⟹ *map-option exInl* (*fun-at s p*) = *gfun-at* (*gterm-of-term s*) *p*
proof (*induct p arbitrary*: *s*)
case *Nil* **then show** *?case*
by (*cases s*) *auto*
next
case (*Cons i p*) **then show** *?case*
by (*cases s*) *auto*
qed

lemmas *gfun-at-gterm-of-term'* [*simp*] = *gfun-at-gterm-of-term*[*OF ground-term-of-gterm*, *unfolded term-of-gterm-inv*]

lemma *gfun-at-None-ngposs-iff*: *gfun-at s p* = *None* ⟷ *p* ∉ *gposs s*
by (*induct rule*: *gfun-at.induct*) *auto*

lemma *gfun-at-map-gterm* [*simp*]:
gfun-at (*map-gterm f t*) *p* = *map-option f* (*gfun-at t p*)
by (*induct t arbitrary*: *p*; *case-tac p*) (*auto simp*: *comp-def*)

lemma *set-gterm-gposs-conv*:
set-gterm t = {*the* (*gfun-at t p*) | *p*. *p* ∈ *gposs t*}
proof (*induct t*)
case (*GFun f ts*)
note [*simp*] = *gfun-at-gterm-of-term*[*OF ground-term-of-gterm*, *unfolded term-of-gterm-inv*]
have [*simp*]: {*the* (*map-option exInl* (*fun-at* (*Fun f* (*map term-of-gterm ts* :: (-, *unit*) *term list*)) *p*)) |*p*.
∃ *i pa*. *p* = *i* # *pa* ∧ *i* < *length ts* ∧ *pa* ∈ *gposs* (*ts* ! *i*)} =
(⋃ *x*∈{*ts* ! *i* |*i*. *i* < *length ts*}. {*the* (*gfun-at x p*) |*p*. *p* ∈ *gposs x*})
unfolding *UNION-eq*
proof ((*intro set-eqI iffI*; *elim CollectE exE bexE conjE*), *goal-cases lr rl*)
case (*lr x p i pa*) **then show** *?case*
by (*intro CollectI*[*of* - *x*] *bexI*[*of* - *ts* ! *i*] *exI*[*of* - *pa*]) (*auto intro*!: *arg-cong*[**where** *?f* = *the*])
next
case (*rl x xa i p*) **then show** *?case*

by (*intro CollectI*[*of* - *x*] *exI*[*of* - *i* # *p*]) *auto*
qed
have [*simp*]: (⋃ *x*∈{*ts* ! *i* |*i*. *i* < *length ts*}. {*the* (*gfun-at x p*) |*p*. *p* ∈ *gposs x*}) =
{*the* (*gfun-at* (*GFun f ts*) *p*) |*p*. ∃ *i pa*. *p* = *i* # *pa* ∧ *i* < *length ts* ∧ *pa* ∈ *gposs* (*ts* ! *i*)}
by *auto* (*metis gfun-at.simps*(*2*))+
show *?case*
by (*simp add*: *GFun*(*1*) *set-conv-nth conj-disj-distribL ex-disj-distrib Collect-disj-eq*)

qed

A *gterm* version of lemma `eq_term_by_poss_fun_at`.

lemma *fun-at-gfun-at-conv*:
fun-at (*term-of-gterm s*) *p* = *fun-at* (*term-of-gterm t*) *p* ⟷ *gfun-at s p* = *gfun-at t p*
proof (*induct p arbitrary*: *s t*)
case *Nil* **then show** *?case*
by (*cases s*; *cases t*) *auto*
next
case (*Cons i p*)
obtain *f h ss ts* **where** [*simp*]: *s* = *GFun f ss t* = *GFun h ts* **by** (*cases s*; *cases t*) *auto*
have [*simp*]: *None* = *fun-at* (*term-of-gterm* (*ts* ! *i*)) *p* ⟷ *p* ∉ *gposs* (*ts* ! *i*)
using *fun-at-None-nposs-iff* **by** (*metis poss-gposs-mem-conv*)
have [*simp*]:*None* = *gfun-at* (*ts* ! *i*) *p* ⟷ *p* ∉ *gposs* (*ts* ! *i*)
using *gfun-at-None-ngposs-iff* **by** *force*
show *?case* **using** *Cons*[*of gargs s* ! *i gargs t* ! *i*]
by (*auto simp*: *poss-gposs-conv gfun-at-None-ngposs-iff fun-at-None-nposs-iff*
intro!: *iffD2*[*OF gfun-at-None-ngposs-iff*] *iffD2*[*OF fun-at-None-nposs-iff*])
qed

lemmas *eq-gterm-by-gposs-gfun-at* = *arg-cong*[**where** *f* = *gterm-of-term*,
OF eq-term-by-poss-fun-at[*of term-of-gterm s* :: (-, *unit*) *term term-of-gterm t* :: (-, *unit*) *term* **for** *s t*],
unfolded term-of-gterm-inv poss-gposs-conv fun-at-gfun-at-conv]

fun *gsubt-at* :: *'f gterm* ⇒ *pos* ⇒ *'f gterm* **where**
gsubt-at s [] = *s* |
gsubt-at (*GFun f ss*) (*i* # *p*) = *gsubt-at* (*ss* ! *i*) *p*

lemma *gsubt-at-to-subt-at*:
assumes *p* ∈ *gposs s*
shows *gterm-of-term* (*term-of-gterm s* |- *p*) = *gsubt-at s p*
using *assms* **by** (*induct arbitrary*: *p*) (*auto simp add*: *map-idI*)

lemma *term-of-gterm-gsubt*:
assumes *p* ∈ *gposs s*
shows (*term-of-gterm s*) |- *p* = *term-of-gterm* (*gsubt-at s p*)

using *assms* **by** (*induct arbitrary*: *p*) *auto*

lemma *gsubt-at-gposs* [*simp*]:
assumes $p \in gposs\ s$
shows $gposs\ (gsubt\text{-}at\ s\ p) = \{x \mid x.\ p\ @\ x \in gposs\ s\}$
using *assms* **by** (*induct s arbitrary*: *p*) *auto*

lemma *gfun-at-gsub-at* [*simp*]:
assumes $p \in gposs\ s$ **and** $p\ @\ q \in gposs\ s$
shows $gfun\text{-}at\ (gsubt\text{-}at\ s\ p)\ q = gfun\text{-}at\ s\ (p\ @\ q)$
using *assms* **by** (*induct s arbitrary*: *p q*) *auto*

lemma *gposs-gsubst-at-subst-at-eq* [*simp*]:
assumes $p \in gposs\ s$
shows $gposs\ (gsubt\text{-}at\ s\ p) = poss\ (term\text{-}of\text{-}gterm\ s\ |\text{-}\ p)$ **using** *assms*
proof (*induct s arbitrary*: *p*)
case (*GFun f ts*)
show *?case* **using** *GFun*(*1*)[*OF nth-mem*] *GFun*(*2*−)
by (*auto simp*: *poss-gposs-mem-conv*) *blast*+
qed

lemma *gpos-append-gposs*:
assumes $p \in gposs\ t$ **and** $q \in gposs\ (gsubt\text{-}at\ t\ p)$
shows $p\ @\ q \in gposs\ t$
using *assms* **by** *auto*

Replace terms at position

fun *replace-gterm-at* ($\langle$-[- $\leftarrow$ -]$_G\rangle$ [*1000*, *0*, *0*] *1000*) **where**
replace-gterm-at s [] *t* = *t*
| *replace-gterm-at* (*GFun f ts*) (*i* # *ps*) *t* =
(*if i* < *length ts then GFun f* (*ts*[*i*:=(*replace-gterm-at* (*ts* ! *i*) *ps t*)]) *else GFun f ts*)

lemma *replace-gterm-at-not-poss* [*simp*]:
$p \notin gposs\ s \Longrightarrow s[p \leftarrow t]_G = s$
proof (*induct s arbitrary*: *p*)
case (*GFun f ts*) **show** *?case* **using** *GFun*(*1*)[*OF nth-mem*] *GFun*(*2*)
by (*cases p*) (*auto simp*: *min-def intro*!: *nth-equalityI*)
qed

lemma *parallel-replace-gterm-commute* [*ac-simps*]:
$p \perp q \Longrightarrow s[p \leftarrow t]_G[q \leftarrow u]_G = s[q \leftarrow u]_G[p \leftarrow t]_G$
proof (*induct s arbitrary*: *p q*)
case (*GFun f ts*)
from *GFun*(*2*) **have** $p \neq []\ q \neq []$ **by** *auto*
then obtain *i j ps qs* **where** [*simp*]: $p = i \# ps\ q = j \# qs$
by (*cases p*; *cases q*) *auto*
have $i \neq j \Longrightarrow (GFun\ f\ ts)[p \leftarrow t]_G[q \leftarrow u]_G = (GFun\ f\ ts)[q \leftarrow u]_G[p \leftarrow t]_G$
by (*auto simp*: *list-update-swap*)

then show *?case* **using** *GFun(1)*[*OF nth-mem*, *of j ps qs*] *GFun(2)*
by (*cases i = j*) (*auto simp*: *par-Cons-iff*)
qed

lemma *replace-gterm-at-above* [*simp*]:
$p \leq_p q \Longrightarrow s[q \leftarrow t]_G[p \leftarrow u]_G = s[p \leftarrow u]_G$
proof (*induct p arbitrary*: *s q*)
case (*Cons i p*)
show *?case* **using** *Cons(1)*[*of tl q gargs s* ! *i*] *Cons(2)*
by (*cases q*; *cases s*) *auto*
qed *auto*

lemma *replace-gterm-at-below* [*simp*]:
$p <_p q \Longrightarrow s[p \leftarrow t]_G[q \leftarrow u]_G = s[p \leftarrow t[q -_p p \leftarrow u]_G]_G$
proof (*induct p arbitrary*: *s q*)
case (*Cons i p*)
show *?case* **using** *Cons(1)*[*of tl q gargs s* ! *i*] *Cons(2)*
by (*cases q*; *cases s*) *auto*
qed *auto*

lemma *groot-sym-replace-gterm* [*simp*]:
$p \neq [] \Longrightarrow$ *groot-sym* $s[p \leftarrow t]_G =$ *groot-sym s*
by (*cases s*; *cases p*) *auto*

lemma *replace-gterm-gsubt-at-id* [*simp*]: $s[p \leftarrow$ *gsubt-at s p*$]_G = s$
proof (*induct p arbitrary*: *s*)
case (*Cons i p*) **then show** *?case*
by (*cases s*) *auto*
qed *auto*

lemma *replace-gterm-conv*:
$p \in$ *gpos s* $\Longrightarrow$ (*term-of-gterm s*)$[p \leftarrow$ (*term-of-gterm t*)$]$ = *term-of-gterm* ($s[p \leftarrow t]_G$)
proof (*induct p arbitrary*: *s*)
case (*Cons i p*) **then show** *?case*
by (*cases s*) (*auto simp*: *nth-list-update intro*: *nth-equalityI*)
qed *auto*

2.9.2 Tree domains

type-synonym *gdomain* = *unit gterm*

abbreviation *gdomain* **where**
gdomain ≡ *map-gterm* (λ-. ())

lemma *gdomain-id*:
gdomain t = *t*
proof −
have [*simp*]: (λ-. ()) = *id* **by** *auto*

then show *?thesis* **by** (*simp add*: *gterm.map-id*)
qed

lemma *gdomain-gsubt* [*simp*]:
assumes $p \in gposs\ t$
shows $gdomain\ (gsubt\text{-}at\ t\ p) = gsubt\text{-}at\ (gdomain\ t)\ p$
using *assms* **by** (*induct t arbitrary*: *p*) *auto*

Union of tree domains

fun *gunion* :: *gdomain* ⇒ *gdomain* ⇒ *gdomain* **where**
gunion (*GFun f ss*) (*GFun g ts*) = *GFun* () (*map* (λ*i*.
if i < *length ss then if i* < *length ts then gunion* (*ss* ! *i*) (*ts* ! *i*)
else ss ! *i else ts* ! *i*) [*0*..<*max* (*length ss*) (*length ts*)])

lemma *gposs-gunion* [*simp*]:
gposs (*gunion s t*) = *gposs s* ∪ *gposs t*
by (*induct s t rule*: *gunion.induct*) (*auto simp*: *less-max-iff-disj split*: *if-splits*)

lemma *gunion-unit* [*simp*]:
gunion s (*GFun* () []) = *s gunion* (*GFun* () []) *s* = *s*
by (*cases s*, (*auto intro*!: *nth-equalityI*)[*1*])+

lemma *gunion-gsubt-at-nt-poss1*:
assumes $p \in gposs\ s$ **and** $p \notin gposs\ t$
shows $gsubt\text{-}at\ (gunion\ s\ t)\ p = gsubt\text{-}at\ s\ p$
using *assms* **by** (*induct s arbitrary*: *p t*) (*case-tac p*; *case-tac t*, *auto*)

lemma *gunion-gsubt-at-nt-poss2*:
assumes $p \in gposs\ t$ **and** $p \notin gposs\ s$
shows $gsubt\text{-}at\ (gunion\ s\ t)\ p = gsubt\text{-}at\ t\ p$
using *assms* **by** (*induct t arbitrary*: *p s*) (*case-tac p*; *case-tac s*, *auto*)

lemma *gunion-gsubt-at-poss*:
assumes $p \in gposs\ s$ **and** $p \in gposs\ t$
shows $gunion\ (gsubt\text{-}at\ s\ p)\ (gsubt\text{-}at\ t\ p) = gsubt\text{-}at\ (gunion\ s\ t)\ p$
using *assms*
proof (*induct p arbitrary*: *s t*)
case (*Cons a p*)
then show *?case* **by** (*cases s*; *cases t*) *auto*
qed *auto*

lemma *gfun-at-domain*:
shows *gfun-at t p* = (*if p* ∈ *gposs t then Some* () *else None*)
proof (*induct t arbitrary*: *p*)
case (*GFun f ts*) **then show** *?case*
by (*cases p*) *auto*
qed

lemma *gunion-assoc* [*ac-simps*]:
gunion s (*gunion t u*) = *gunion* (*gunion s t*) *u*
by (*intro eq-gterm-by-gposs-gfun-at*) (*auto simp*: *gfun-at-domain poss-gposs-mem-conv*)

lemma *gunion-commute* [*ac-simps*]:
gunion s t = *gunion t s*
by (*intro eq-gterm-by-gposs-gfun-at*) (*auto simp*: *gfun-at-domain poss-gposs-mem-conv*)

lemma *gunion-idemp* [*simp*]:
gunion s s = *s*
by (*intro eq-gterm-by-gposs-gfun-at*) (*auto simp*: *gfun-at-domain poss-gposs-mem-conv*)

definition *gunions* :: *gdomain list* ⇒ *gdomain* **where**
gunions ts = *foldr gunion ts* (*GFun* () [])

lemma *gunions-append*:
gunions (*ss* @ *ts*) = *gunion* (*gunions ss*) (*gunions ts*)
by (*induct ss*) (*auto simp*: *gunions-def gunion-assoc*)

lemma *gposs-gunions* [*simp*]:
gposs (*gunions ts*) = {[]} ∪ ⋃{*gposs t* |*t*. *t* ∈ *set ts*}
by (*induct ts*) (*auto simp*: *gunions-def*)

Given a tree domain and a function from positions to symbols, we can construct a term.

context
notes *conj-cong* [*fundef-cong*]
begin
fun *glabel* :: (*pos* ⇒ *'f*) ⇒ *gdomain* ⇒ *'f gterm* **where**
glabel h (*GFun f ts*) = *GFun* (*h* []) (*map* (λ*i*. *glabel* (*h* ∘ (#) *i*) (*ts* ! *i*)) [*0*..<*length ts*])
end

lemma *map-gterm-glabel*:
map-gterm f (*glabel h t*) = *glabel* (*f* ∘ *h*) *t*
by (*induct t arbitrary*: *h*) (*auto simp*: *comp-def*)

lemma *gfun-at-glabel* [*simp*]:
gfun-at (*glabel f t*) *p* = (*if p* ∈ *gposs t then Some* (*f p*) *else None*)
by (*induct t arbitrary*: *f p*, *case-tac p*) (*auto simp*: *comp-def*)

lemma *gposs-glabel* [*simp*]:
gposs (*glabel f t*) = *gposs t*
by (*induct t arbitrary*: *f*) *auto*

lemma *glabel-map-gterm-conv*:
glabel (*f* ∘ *gfun-at t*) (*gdomain t*) = *map-gterm* (*f* ∘ *Some*) *t*
by (*induct t*) (*auto simp*: *comp-def intro*!: *nth-equalityI*)

lemma *gfun-at-nongposs* [*simp*]:
p ∉ *gposs t* ⟹ *gfun-at t p* = *None*
using *gfun-at-glabel*[*of the* ∘ *gfun-at t gdomain t p*, *unfolded glabel-map-gterm-conv*]
by (*simp add*: *comp-def option.map-ident*)

lemma *gfun-at-poss*:
p ∈ *gposs t* ⟹ ∃*f*. *gfun-at t p* = *Some f*
using *gfun-at-glabel*[*of the* ∘ *gfun-at t gdomain t p*, *unfolded glabel-map-gterm-conv*]
by (*auto simp*: *comp-def*)

lemma *gfun-at-possE*:
assumes *p* ∈ *gposs t*
obtains *f* **where** *gfun-at t p* = *Some f*
using *assms gfun-at-poss* **by** *blast*

lemma *gfun-at-poss-gpossD*:
gfun-at t p = *Some f* ⟹ *p* ∈ *gposs t*
by (*metis gfun-at-nongposs option.distinct*(*1*))

function symbols of a ground term

primrec *funas-gterm* :: *'f gterm* ⇒ (*'f* × *nat*) *set* **where**
funas-gterm (*GFun f ts*) = {(*f*, *length ts*)} ∪ ⋃(*set* (*map funas-gterm ts*))

lemma *funas-gterm-gterm-of-term*:
ground t ⟹ *funas-gterm* (*gterm-of-term t*) = *funas-term t*
by (*induct t*) (*auto simp*: *funas-gterm-def*)

lemma *funas-term-of-gterm-conv*:
funas-term (*term-of-gterm t*) = *funas-gterm t*
by (*induct t*) (*auto simp*: *funas-gterm-def*)

lemma *funas-gterm-map-gterm*:
assumes *funas-gterm t* ⊆ *F*
shows *funas-gterm* (*map-gterm f t*) ⊆ (λ (*h*, *n*). (*f h*, *n*)) ‘ *F*
using *assms* **by** (*induct t*) (*auto simp*: *funas-gterm-def*)

lemma *gterm-of-term-inj*:
assumes ⋀ *t*. *t* ∈ *S* ⟹ *ground t*
shows *inj-on gterm-of-term S*
using *assms gterm-of-term-inv* **by** (*fastforce simp*: *inj-on-def*)

lemma *funas-gterm-gsubt-at-subseteq*:
assumes *p* ∈ *gposs s*
shows *funas-gterm* (*gsubt-at s p*) ⊆ *funas-gterm s* **using** *assms*
apply (*induct s arbitrary*: *p*) **apply** *auto*
using *nth-mem* **by** *blast*+

lemma *finite-funas-gterm*: *finite* (*funas-gterm t*)
by (*induct t*) *auto*

ground term set

abbreviation *gterms* **where**
gterms $\mathcal{F} \equiv \{s.$ *funas-gterm* $s \subseteq \mathcal{F}\}$

lemma *gterms-mono*:
$\mathcal{G} \subseteq \mathcal{F} \Longrightarrow$ *gterms* $\mathcal{G} \subseteq$ *gterms* $\mathcal{F}$
by *auto*

inductive-set $\mathcal{T}_G$ **for** $\mathcal{F}$ **where**
const [*simp*]: $(a, 0) \in \mathcal{F} \Longrightarrow$ *GFun* a $[] \in \mathcal{T}_G$ $\mathcal{F}$
| *ind* [*intro*]: $(f, n) \in \mathcal{F} \Longrightarrow$ *length* $ss = n \Longrightarrow (\bigwedge i.\ i <$ *length* $ss \Longrightarrow ss$! $i \in \mathcal{T}_G$ $\mathcal{F}) \Longrightarrow$ *GFun* f $ss \in \mathcal{T}_G$ $\mathcal{F}$

lemma $\mathcal{T}_G$*-sound*:
$s \in \mathcal{T}_G$ $\mathcal{F} \Longrightarrow$ *funas-gterm* $s \subseteq \mathcal{F}$
proof (*induct*)
case (*GFun* f *ts*)
show *?case* **using** *GFun(1)*[*OF nth-mem*] *GFun(2)*
by (*fastforce simp*: *in-set-conv-nth elim*!: $\mathcal{T}_G$*.cases intro*: *nth-mem*)
qed

lemma $\mathcal{T}_G$*-complete*:
funas-gterm $s \subseteq \mathcal{F} \Longrightarrow s \in \mathcal{T}_G$ $\mathcal{F}$
by (*induct* s) (*auto simp*: *SUP-le-iff*)

lemma $\mathcal{T}_G$*-funas-gterm-conv*:
$s \in \mathcal{T}_G$ $\mathcal{F} \longleftrightarrow$ *funas-gterm* $s \subseteq \mathcal{F}$
using $\mathcal{T}_G$*-sound* $\mathcal{T}_G$*-complete* **by** *auto*

lemma $\mathcal{T}_G$*-equivalent-def*:
$\mathcal{T}_G$ $\mathcal{F}$ = *gterms* $\mathcal{F}$
using $\mathcal{T}_G$*-funas-gterm-conv* **by** *auto*

lemma $\mathcal{T}_G$*-intersection* [*simp*]:
$s \in \mathcal{T}_G$ $\mathcal{F} \Longrightarrow s \in \mathcal{T}_G$ $\mathcal{G} \Longrightarrow s \in \mathcal{T}_G$ $(\mathcal{F} \cap \mathcal{G})$
by (*auto simp*: $\mathcal{T}_G$*-funas-gterm-conv* $\mathcal{T}_G$*-equivalent-def*)

lemma $\mathcal{T}_G$*-mono*:
$\mathcal{G} \subseteq \mathcal{F} \Longrightarrow \mathcal{T}_G$ $\mathcal{G} \subseteq \mathcal{T}_G$ $\mathcal{F}$
using *gterms-mono* **by** (*simp add*: $\mathcal{T}_G$*-equivalent-def*)

lemma $\mathcal{T}_G$*-UNIV* [*simp*]: $s \in \mathcal{T}_G$ *UNIV*
by (*induct*) *auto*

definition *funas-grel* **where**
funas-grel $\mathcal{R} = \bigcup$ $((\lambda\ (s, t).$ *funas-gterm* $s \cup$ *funas-gterm* $t)$ ‘ $\mathcal{R})$

end
theory *FSet-Utils*

imports *HOL−Library.FSet*
HOL−Library.List-Lexorder
Ground-Terms
begin

context
includes *fset.lifting*
begin

lift-definition *fCollect* :: (*'a* ⇒ *bool*) ⇒ *'a fset* **is** λ *P*. *if finite* (*Collect P*) *then Collect P else* {}
by *auto*

lift-definition *fSigma* :: *'a fset* ⇒ (*'a* ⇒ *'b fset*) ⇒ (*'a* × *'b*) *fset* **is** *Sigma*
by *auto*

lift-definition *is-fempty* :: *'a fset* ⇒ *bool* **is** *Set.is-empty* **.**
lift-definition *fremove* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **is** *Set.remove*
by (*simp add*: *remove-def*)

lift-definition *finj-on* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *bool* **is** *inj-on* **.**
lift-definition *the-finv-into* :: *'a fset* ⇒ (*'a* ⇒ *'b*) ⇒ *'b* ⇒ *'a* **is** *the-inv-into* **.**

lemma *fCollect-memberI* [*intro*!]:
finite (*Collect P*) ⟹ *P x* ⟹ *x* |∈| *fCollect P*
by *transfer auto*

lemma *fCollect-member* [*iff*]:
x |∈| *fCollect P* ⟷ *finite* (*Collect P*) ∧ *P x*
by *transfer* (*auto split*: *if-splits*)

lemma *fCollect-cong*: (⋀*x*. *P x* = *Q x*) ⟹ *fCollect P* = *fCollect Q*
by *presburger*
end

syntax
-fColl :: *pttrn* ⇒ *bool* ⇒ *'a set* (‹(*1*{|-./ -|})›)
syntax-consts
-fColl ⇌ *fCollect*
translations
{|*x*. *P*|} ⇌ *CONST fCollect* (λ*x*. *P*)

syntax (*ASCII*)
-fCollect :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *'a set* (‹(*1*{(-/|:| -)./ -})›)
syntax
-fCollect :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *'a set* (‹(*1*{(-/ |∈| -)./ -})›)
syntax-consts
-fCollect ⇌ *fCollect*
translations

```
  {p|:|A. P} ⇀ CONST fCollect (λp. p |∈| A ∧ P)

syntax
  -fSetcompr :: 'a ⇒ idts ⇒ bool ⇒ 'a fset    (‹(1{|- |/-./ -|})›)

syntax-consts
  -fSetcompr ⇌ fCollect

parse-translation ‹
  let
    val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX , const-syntax‹Ex›));

    fun nvars (Const (syntax-const‹-idts›, -) $ - $ idts) = nvars idts + 1
      | nvars - = 1;

    fun setcompr-tr ctxt [e, idts, b] =
      let
        val eq = Syntax.const const-syntax‹HOL.eq› $ Bound (nvars idts) $ e;
        val P = Syntax.const const-syntax‹HOL.conj› $ eq $ b;
        val exP = ex-tr ctxt [idts, P];
      in Syntax.const const-syntax‹fCollect› $ absdummy dummyT exP end;

  in [(syntax-const‹-fSetcompr›, setcompr-tr)] end
›

print-translation ‹
let
  val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (const-syntax‹Ex›, DUMMY));

  fun setcompr-tr' ctxt [Abs (abs as (-, -, P))] =
    let
      fun check (Const (const-syntax‹Ex›, -) $ Abs (-, -, P), n) = check (P, n +
1)
        | check (Const (const-syntax‹HOL.conj›, -) $
              (Const (const-syntax‹HOL.eq›, -) $ Bound m $ e) $ P, n) =
            n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
            subset (=) (0 upto (n − 1), add-loose-bnos (e, 0, []))
        | check - = false;

        fun tr' (- $ abs) =
          let val - $ idts $ (- $ (- $ - $ e) $ Q) = ex-tr' ctxt [abs]
          in Syntax.const syntax-const‹-fSetcompr› $ e $ idts $ Q end;
    in
      if check (P, 0) then tr' P
      else
        let
          val (x as - $ Free(xN, -), t) = Syntax-Trans.atomic-abs-tr' ctxt abs;
          val M = Syntax.const syntax-const‹-fColl› $ x $ t;
        in
```

```
        case t of
          Const (const-syntax‹HOL.conj›, -) $
            (Const (const-syntax‹fmember›, -) $
              (Const (syntax-const‹-bound›, -) $ Free (yN, -)) $ A) $ P =>
          if xN = yN then Syntax.const syntax-const‹-fCollect› $ x $ A $ P else
M
        | - => M
      end
    end;
  in [(const-syntax‹fCollect›, setcompr-tr')] end
›

syntax
  -fSigma :: pttrn ⇒ 'a fset ⇒ 'b fset ⇒ ('a × 'b) set  (‹(3fSIGMA -|:|-./ -)› [0,
0, 10] 10)
syntax-consts
  -fSigma ⇌ fSigma
translations
  fSIGMA x|:|A. B ⇌ CONST fSigma A (λx. B)

notation
  ffUnion (‹|⋃|›)

context
includes fset.lifting
begin

lemma right-total-cr-fset [transfer-rule]:
  right-total cr-fset
  by (auto simp: cr-fset-def right-total-def)

lemma bi-unique-cr-fset [transfer-rule]:
  bi-unique cr-fset
  by (auto simp: bi-unique-def cr-fset-def fset-inject)

lemma right-total-pcr-fset-eq [transfer-rule]:
  right-total (pcr-fset (=))
  by (simp add: right-total-cr-fset fset.pcr-cr-eq)

lemma bi-unique-pcr-fset [transfer-rule]:
  bi-unique (pcr-fset (=))
  by (simp add: fset.pcr-cr-eq bi-unique-cr-fset)


lemma set-fset-of-list-transfer [transfer-rule]:
  rel-fun (list-all2 A) (pcr-fset A) set fset-of-list
  unfolding pcr-fset-def rel-set-def rel-fun-def
  by (force simp: list-all2-conv-all-nth in-set-conv-nth cr-fset-def fset-of-list.rep-eq
relcompp-apply)
```

lemma *fCollectD*: $a \mid\in\mid \{\mid x\ .\ P\ x\mid\} \Longrightarrow P\ a$
by *transfer* (*auto split*: *if-splits*)

lemma *fCollectI*: $P\ a \Longrightarrow$ *finite* (*Collect* P) $\Longrightarrow a \mid\in\mid \{\mid x.\ P\ x\mid\}$
by (*auto intro*: *fCollect-memberI*)

lemma *fCollect-fempty-eq* [*simp*]: *fCollect* $P = \{\mid\mid\} \longleftrightarrow (\forall x.\ \neg\ P\ x) \vee$ *infinite* (*Collect* P)
by *auto*

lemma *fempty-fCollect-eq* [*simp*]: $\{\mid\mid\} =$ *fCollect* $P \longleftrightarrow (\forall x.\ \neg\ P\ x) \vee$ *infinite* (*Collect* P)
by *auto*

lemma *fset-image-conv*:
$\{f\ x \mid x.\ x \mid\in\mid T\} =$ *fset* ($f\ \mid`\mid\ T$)
by *transfer auto*

lemma *fimage-def*:
$f\ \mid`\mid\ A = \{\mid y.\ \exists x \mid\in\mid A.\ y = f\ x\ \mid\}$
by *transfer auto*

lemma *ffilter-simp*: *ffilter* $P\ A = \{a \mid\in\mid A.\ P\ a\}$
by *transfer auto*

lemmas *fset-list-fsubset-eq-nth-conv* = *set-list-subset-eq-nth-conv*[*Transfer.transferred*]
lemmas *mem-idx-fset-sound* = *mem-idx-sound*[*Transfer.transferred*]
— Dealing with fset products

abbreviation *fTimes* :: $'a$ *fset* $\Rightarrow$ $'b$ *fset* $\Rightarrow$ ($'a \times 'b$) *fset* (**infixr** $\langle\mid\times\mid\rangle$ *80*)
where $A \mid\times\mid B \equiv$ *fSigma* A (λ-. B)

lemma *fSigma-repeq*:
fset ($A \mid\times\mid B$) = *fset* $A \times$ *fset* B
by (*transfer*) *auto*

lemmas *fSigmaI* [*intro*!] = *SigmaI*[*Transfer.transferred*]
lemmas *fSigmaE* [*elim*!] = *SigmaE*[*Transfer.transferred*]
lemmas *fSigmaD1* = *SigmaD1*[*Transfer.transferred*]
lemmas *fSigmaD2* = *SigmaD2*[*Transfer.transferred*]
lemmas *fSigmaE2* = *SigmaE2*[*Transfer.transferred*]
lemmas *fSigma-cong* = *Sigma-cong*[*Transfer.transferred*]
lemmas *fSigma-mono* = *Sigma-mono*[*Transfer.transferred*]
lemmas *fSigma-empty1* [*simp*] = *Sigma-empty1*[*Transfer.transferred*]
lemmas *fSigma-empty2* [*simp*] = *Sigma-empty2*[*Transfer.transferred*]
lemmas *fmem-Sigma-iff* [*iff*] = *mem-Sigma-iff*[*Transfer.transferred*]

lemmas *fmem-Times-iff* = *mem-Times-iff*[*Transfer.transferred*]
lemmas *fSigma-empty-iff* = *Sigma-empty-iff*[*Transfer.transferred*]
lemmas *fTimes-subset-cancel2* = *Times-subset-cancel2*[*Transfer.transferred*]
lemmas *fTimes-eq-cancel2* = *Times-eq-cancel2*[*Transfer.transferred*]
lemmas *fUN-Times-distrib* = *UN-Times-distrib*[*Transfer.transferred*]
lemmas *fsplit-paired-Ball-Sigma* [*simp*, *no-atp*] = *split-paired-Ball-Sigma*[*Transfer.transferred*]
lemmas *fsplit-paired-Bex-Sigma* [*simp*, *no-atp*] = *split-paired-Bex-Sigma*[*Transfer.transferred*]
lemmas *fSigma-Un-distrib1* = *Sigma-Un-distrib1*[*Transfer.transferred*]
lemmas *fSigma-Un-distrib2* = *Sigma-Un-distrib2*[*Transfer.transferred*]
lemmas *fSigma-Int-distrib1* = *Sigma-Int-distrib1*[*Transfer.transferred*]
lemmas *fSigma-Int-distrib2* = *Sigma-Int-distrib2*[*Transfer.transferred*]
lemmas *fSigma-Diff-distrib1* = *Sigma-Diff-distrib1*[*Transfer.transferred*]
lemmas *fSigma-Diff-distrib2* = *Sigma-Diff-distrib2*[*Transfer.transferred*]
lemmas *fSigma-Union* = *Sigma-Union*[*Transfer.transferred*]
lemmas *fTimes-Un-distrib1* = *Times-Un-distrib1*[*Transfer.transferred*]
lemmas *fTimes-Int-distrib1* = *Times-Int-distrib1*[*Transfer.transferred*]
lemmas *fTimes-Diff-distrib1* = *Times-Diff-distrib1*[*Transfer.transferred*]
lemmas *fTimes-empty* [*simp*] = *Times-empty*[*Transfer.transferred*]
lemmas *ftimes-subset-iff* = *times-subset-iff*[*Transfer.transferred*]
lemmas *ftimes-eq-iff* = *times-eq-iff*[*Transfer.transferred*]
lemmas *ffst-image-times* [*simp*] = *fst-image-times*[*Transfer.transferred*]
lemmas *fsnd-image-times* [*simp*] = *snd-image-times*[*Transfer.transferred*]
lemmas *fsnd-image-Sigma* = *snd-image-Sigma*[*Transfer.transferred*]
lemmas *finsert-Times-insert* = *insert-Times-insert*[*Transfer.transferred*]
lemmas *fTimes-Int-Times* = *Times-Int-Times*[*Transfer.transferred*]
lemmas *fimage-paired-Times* = *image-paired-Times*[*Transfer.transferred*]
lemmas *fproduct-swap* = *product-swap*[*Transfer.transferred*]
lemmas *fswap-product* = *swap-product*[*Transfer.transferred*]
lemmas *fsubset-fst-snd* = *subset-fst-snd*[*Transfer.transferred*]
lemmas *map-prod-ftimes* = *map-prod-times*[*Transfer.transferred*]

lemma *fCollect-case-prod* [*simp*]:
  {|(*a*, *b*). *P a* ∧ *Q b*|} = *fCollect P* |×| *fCollect Q*
  by *transfer* (*auto dest*: *finite-cartesian-productD1 finite-cartesian-productD2*)
lemma *fCollect-case-prodD*:
  x |∈| {|(*x*, *y*). *A x y*|} ⟹ *A* (*fst x*) (*snd x*)
  by *auto*

lemmas *fCollect-case-prod-Sigma* = *Collect-case-prod-Sigma*[*Transfer.transferred*]
lemmas *ffst-image-Sigma* = *fst-image-Sigma*[*Transfer.transferred*]
lemmas *fimage-split-eq-Sigma* = *image-split-eq-Sigma*[*Transfer.transferred*]

— Dealing with transitive closure

lift-definition *ftrancl* :: ('*a* × '*a*) *fset* ⇒ ('*a* × '*a*) *fset* (‹(-|⁺|)› [*1000*] *999*) **is**

trancl
by *auto*

lemmas *fr-into-trancl* [*intro*, *Pure.intro*] = *r-into-trancl*[*Transfer.transferred*]
lemmas *ftrancl-into-trancl* [*Pure.intro*] = *trancl-into-trancl*[*Transfer.transferred*]
lemmas *ftrancl-induct*[*consumes 1*, *case-names Base Step*] = *trancl.induct*[*Transfer.transferred*]
lemmas *ftrancl-mono* = *trancl-mono*[*Transfer.transferred*]
lemmas *ftrancl-trans*[*trans*] = *trancl-trans*[*Transfer.transferred*]
lemmas *ftrancl-empty* [*simp*] = *trancl-empty* [*Transfer.transferred*]
lemmas *ftranclE*[*cases set*: *ftrancl*] = *tranclE*[*Transfer.transferred*]
lemmas *converse-ftrancl-induct*[*consumes 1*, *case-names Base Step*] = *converse-trancl-induct*[*Transfer.transfe*
lemmas *converse-ftranclE* = *converse-tranclE*[*Transfer.transferred*]
lemma *in-ftrancl-UnI*:
x |∈| *R*|^+^| ∨ *x* |∈| *S*|^+^| ⟹ *x* |∈| (*R* |∪| *S*)|^+^|
by *transfer* (*auto simp add*: *trancl-mono*)

lemma *ftranclD*:
(*x*, *y*) |∈| *R*|^+^| ⟹ ∃ *z*. (*x*, *z*) |∈| *R* ∧ (*z* = *y* ∨ (*z*, *y*) |∈| *R*|^+^|)
by (*induct rule*: *ftrancl-induct*) (*auto*, *meson ftrancl-into-trancl*)

lemma *ftranclD2*:
(*x*, *y*) |∈| *R*|^+^| ⟹ ∃ *z*. (*x* = *z* ∨ (*x*, *z*) |∈| *R*|^+^|) ∧ (*z*, *y*) |∈| *R*
by (*induct rule*: *ftrancl-induct*) *auto*

lemma *not-ftrancl-into*:
(*x*, *z*) |∉| *r*|^+^| ⟹ (*y*, *z*) |∈| *r* ⟹ (*x*, *y*) |∉| *r*|^+^|
by *transfer* (*auto simp add*: *trancl.trancl-into-trancl*)
lemmas *ftrancl-map-both-fRestr* = *trancl-map-both-Restr*[*Transfer.transferred*]
lemma *ftrancl-map-both-fsubset*:
finj-on f X ⟹ *R* |⊆| *X* |×| *X* ⟹ (*map-both f* |‘| *R*)|^+^| = *map-both f* |‘| *R*|^+^|
using *ftrancl-map-both-fRestr*[*of f X R*]
by (*simp add*: *inf-absorb1*)
lemmas *ftrancl-map-prod-mono* = *trancl-map-prod-mono*[*Transfer.transferred*]
lemmas *ftrancl-map* = *trancl-map*[*Transfer.transferred*]

lemmas *ffUnion-iff* [*simp*] = *Union-iff*[*Transfer.transferred*]
lemmas *ffUnionI* [*intro*] = *UnionI*[*Transfer.transferred*]
lemmas *fUn-simps* [*simp*] = *UN-simps*[*Transfer.transferred*]

lemmas *fINT-simps* [*simp*] = *INT-simps*[*Transfer.transferred*]

lemmas *fUN-ball-bex-simps* [*simp*] = *UN-ball-bex-simps*[*Transfer.transferred*]

lemmas *in-fset-conv-nth* = *in-set-conv-nth*[*Transfer.transferred*]
lemmas *fnth-mem* [*simp*] = *nth-mem*[*Transfer.transferred*]

lemmas *distinct-sorted-list-of-fset* = *distinct-sorted-list-of-set* [*Transfer.transferred*]
lemmas *fcard-fset* = *card-set*[*Transfer.transferred*]
lemma *upt-fset*:
fset-of-list [*i*..<*j*] = *fCollect* (λ *n*. *i* ≤ *n* ∧ *n* < *j*)
by (*induct j arbitrary*: *i*) *auto*

abbreviation *fRestr* :: ('*a* × '*a*) *fset* ⇒ '*a fset* ⇒ ('*a* × '*a*) *fset* **where**
fRestr r A ≡ *r* |∩| (*A* |×| *A*)

lift-definition *fId-on* :: '*a fset* ⇒ ('*a* × '*a*) *fset* **is** *Id-on*
using *Id-on-subset-Times finite-subset* **by** *fastforce*

lemmas *fId-on-empty* [*simp*] = *Id-on-empty* [*Transfer.transferred*]
lemmas *fId-on-eqI* = *Id-on-eqI* [*Transfer.transferred*]
lemmas *fId-onI* [*intro*!] = *Id-onI* [*Transfer.transferred*]
lemmas *fId-onE* [*elim*!] = *Id-onE* [*Transfer.transferred*]
lemmas *fId-on-iff* = *Id-on-iff* [*Transfer.transferred*]
lemmas *fId-on-fsubset-fTimes* = *Id-on-subset-Times* [*Transfer.transferred*]

lift-definition *fconverse* :: ('*a* × '*b*) *fset* ⇒ ('*b* × '*a*) *fset* (‹(-|$^{-1}$|)› [*1000*] *999*)
is *converse* **by** *auto*

lemmas *fconverseI* [*sym*] = *converseI* [*Transfer.transferred*]
lemmas *fconverseD* [*sym*] = *converseD* [*Transfer.transferred*]
lemmas *fconverseE* [*elim*!] = *converseE* [*Transfer.transferred*]
lemmas *fconverse-iff* [*iff*] = *converse-iff*[*Transfer.transferred*]
lemmas *fconverse-fconverse* [*simp*] = *converse-converse*[*Transfer.transferred*]
lemmas *fconverse-empty*[*simp*] = *converse-empty*[*Transfer.transferred*]

lemmas *finj-on-def'* = *inj-on-def*[*Transfer.transferred*]
lemmas *fsubset-finj-on* = *subset-inj-on*[*Transfer.transferred*]
lemmas *the-finv-into-f-f* = *the-inv-into-f-f*[*Transfer.transferred*]
lemmas *f-the-finv-into-f* = *f-the-inv-into-f*[*Transfer.transferred*]
lemmas *the-finv-into-into* = *the-inv-into-into*[*Transfer.transferred*]
lemmas *the-finv-into-onto* [*simp*] = *the-inv-into-onto*[*Transfer.transferred*]
lemmas *the-finv-into-f-eq* = *the-inv-into-f-eq*[*Transfer.transferred*]
lemmas *the-finv-into-comp* = *the-inv-into-comp*[*Transfer.transferred*]
lemmas *finj-on-the-finv-into* = *inj-on-the-inv-into* [*Transfer.transferred*]
lemmas *finj-on-fUn* = *inj-on-Un*[*Transfer.transferred*]

lemma *finj-Inl-Inr*:
finj-on Inl A finj-on Inr A
by (*transfer*, *auto*)+

lemma *finj-CInl-CInr*:
finj-on CInl A finj-on CInr A
using *finj-Inl-Inr* **by** *force+*

lemma *finj-Some*:
finj-on Some A
by (*transfer*, *auto*)

lift-definition *fImage* :: ('*a* × '*b*) *fset* ⇒ '*a fset* ⇒ '*b fset* (**infixr** ‹|‘|› *90*) **is** *Image*
using *finite-Image* **by** *force*

lemmas *fImage-iff* = *Image-iff*[*Transfer.transferred*]
lemmas *fImage-singleton-iff* [*iff*] = *Image-singleton-iff*[*Transfer.transferred*]
lemmas *fImageI* [*intro*] = *ImageI*[*Transfer.transferred*]
lemmas *ImageE* [*elim*!] = *ImageE*[*Transfer.transferred*]
lemmas *frev-ImageI* = *rev-ImageI*[*Transfer.transferred*]
lemmas *fImage-empty1* [*simp*] = *Image-empty1*[*Transfer.transferred*]
lemmas *fImage-empty2* [*simp*] = *Image-empty2*[*Transfer.transferred*]
lemmas *fImage-fInt-fsubset* = *Image-Int-subset*[*Transfer.transferred*]
lemmas *fImage-fUn* = *Image-Un*[*Transfer.transferred*]
lemmas *fUn-fImage* = *Un-Image*[*Transfer.transferred*]
lemmas *fImage-fsubset* = *Image-subset*[*Transfer.transferred*]
lemmas *fImage-eq-fUN* = *Image-eq-UN*[*Transfer.transferred*]
lemmas *fImage-mono* = *Image-mono*[*Transfer.transferred*]
lemmas *fImage-fUN* = *Image-UN*[*Transfer.transferred*]
lemmas *fUN-fImage* = *UN-Image*[*Transfer.transferred*]
lemmas *fSigma-fImage* = *Sigma-Image*[*Transfer.transferred*]

lemmas *fImage-singleton* = *Image-singleton*[*Transfer.transferred*]
lemmas *fImage-Id-on* [*simp*] = *Image-Id-on*[*Transfer.transferred*]
lemmas *fImage-Id* [*simp*] = *Image-Id*[*Transfer.transferred*]
lemmas *fImage-fInt-eq* = *Image-Int-eq*[*Transfer.transferred*]
lemmas *fImage-fsubset-eq* = *Image-subset-eq*[*Transfer.transferred*]
lemmas *fImage-fCollect-case-prod* [*simp*] = *Image-Collect-case-prod*[*Transfer.transferred*]
lemmas *fImage-fINT-fsubset* = *Image-INT-subset*[*Transfer.transferred*]

lemmas *term-fset-induct* = *term.induct*[*Transfer.transferred*]
lemmas *fmap-prod-fimageI* = *map-prod-imageI*[*Transfer.transferred*]
lemmas *finj-on-eq-iff* = *inj-on-eq-iff*[*Transfer.transferred*]
lemmas *prod-fun-fimageE* = *prod-fun-imageE*[*Transfer.transferred*]

lemma *rel-set-cr-fset*:
rel-set cr-fset = (λ *A B*. *A* = *fset* ‘ *B*)
proof −

have *rel-set cr-fset A B* $\longleftrightarrow$ *A = fset ‘ B* **for** *A B*
by (*auto simp*: *image-def rel-set-def cr-fset-def*)
then show *?thesis* **by** *blast*
qed
lemma *pcr-fset-cr-fset*:
pcr-fset cr-fset = (λ *x y*. *x = fset* (*fset* |‘| *y*))
unfolding *pcr-fset-def rel-set-cr-fset*
unfolding *cr-fset-def*
by (*auto simp*: *image-def relcompp-apply*)

lemma *sorted-list-of-fset-id*:
sorted-list-of-fset x = sorted-list-of-fset y ⟹ *x = y*
by (*metis sorted-list-of-fset-simps*(*2*))

lemmas *fBall-def* = *Ball-def*[*Transfer.transferred*]
lemmas *fBex-def* = *Bex-def*[*Transfer.transferred*]
lemmas *fCollectE* = *fCollectD* [*elim-format*]
lemma *fCollect-conj-eq*:
finite (*Collect P*) ⟹ *finite* (*Collect Q*) ⟹ {|*x*. *P x* ∧ *Q x*|} = *fCollect P* |∩|
fCollect Q
by *auto*

lemma *finite-ntrancl*:
finite R ⟹ *finite* (*ntrancl n R*)
by (*induct n*) *auto*

lift-definition *nftrancl* :: *nat* ⇒ (*'a* × *'a*) *fset* ⇒ (*'a* × *'a*) *fset* **is** *ntrancl*
by (*intro finite-ntrancl*) *simp*

lift-definition *frelcomp* :: (*'a* × *'b*) *fset* ⇒ (*'b* × *'c*) *fset* ⇒ (*'a* × *'c*) *fset* (**infixr**
‹|*O*|› *75*) **is** *relcomp*
by (*intro finite-relcomp*) *simp*

lemmas *frelcompE*[*elim*!] = *relcompE*[*Transfer.transferred*]
lemmas *frelcompI*[*intro*] = *relcompI*[*Transfer.transferred*]
lemma *fId-on-frelcomp-id*:
fst |‘| *R* |⊆| *S* ⟹ *fId-on S* |*O*| *R* = *R*
by (*auto intro*!: *frelcompI*)
lemma *fId-on-frelcomp-id2*:
snd |‘| *R* |⊆| *S* ⟹ *R* |*O*| *fId-on S* = *R*
by (*auto intro*!: *frelcompI*)

lemmas *fimage-fset* = *image-set*[*Transfer.transferred*]
lemmas *ftrancl-Un2-separatorE* = *trancl-Un2-separatorE*[*Transfer.transferred*]

lemma *finite-funs-term*: *finite* (*funs-term t*) **by** (*induct t*) *auto*
lemma *finite-funas-term*: *finite* (*funas-term t*) **by** (*induct t*) *auto*
lemma *finite-vars-ctxt*: *finite* (*vars-ctxt C*) **by** (*induct C*) *auto*

lift-definition *ffuns-term* :: ('*f*, '*v*) *term* ⇒ '*f fset* **is** *funs-term* **using** *finite-funs-term*
  by *blast*
lift-definition *fvars-term* :: ('*f*, '*v*) *term* ⇒ '*v fset* **is** *vars-term* **by** *simp*
lift-definition *fvars-ctxt* :: ('*f*, '*v*) *ctxt* ⇒ '*v fset* **is** *vars-ctxt* **by** (*simp add*: *finite-vars-ctxt*)

lemmas *fvars-term-ctxt-apply* [*simp*] = *vars-term-ctxt-apply*[*Transfer.transferred*]
lemmas *fvars-term-of-gterm* [*simp*] = *vars-term-of-gterm*[*Transfer.transferred*]
lemmas *ground-fvars-term-empty* [*simp*] = *ground-vars-term-empty*[*Transfer.transferred*]

lemma *ffuns-term-Var* [*simp*]: *ffuns-term* (*Var x*) = {||}
  by *transfer auto*
lemma *fffuns-term-Fun* [*simp*]: *ffuns-term* (*Fun f ts*) = |⋃| (*ffuns-term* |‘| *fset-of-list ts*) |∪| {|*f*|}
  by *transfer auto*

lemma *fvars-term-Var* [*simp*]: *fvars-term* (*Var x*) = {|*x*|}
  by *transfer auto*
lemma *fvars-term-Fun* [*simp*]: *fvars-term* (*Fun f ts*) = |⋃| (*fvars-term* |‘| *fset-of-list ts*)
  by *transfer auto*

lift-definition *ffunas-term* :: ('*f*, '*v*) *term* ⇒ ('*f* × *nat*) *fset* **is** *funas-term*
  by (*simp add*: *finite-funas-term*)
lift-definition *ffunas-gterm* :: '*f gterm* ⇒ ('*f* × *nat*) *fset* **is** *funas-gterm*
  by (*simp add*: *finite-funas-gterm*)

lemmas *ffunas-term-simps* [*simp*] = *funas-term.simps*[*Transfer.transferred*]
lemmas *ffunas-gterm-simps* [*simp*] = *funas-gterm.simps*[*Transfer.transferred*]
lemmas *ffunas-term-of-gterm-conv* = *funas-term-of-gterm-conv*[*Transfer.transferred*]
lemmas *ffunas-gterm-gterm-of-term* = *funas-gterm-gterm-of-term*[*Transfer.transferred*]

lemma *sorted-list-of-fset-fimage-dist*:
  sorted-list-of-fset (*f* |‘| *A*) = *sort* (*remdups* (*map f* (*sorted-list-of-fset A*)))
  by (*auto simp*: *sorted-list-of-fset.rep-eq simp flip*: *sorted-list-of-set-sort-remdups*)

end

lemma *finite-snd* [*intro*]:
  finite S ⟹ *finite* {*x*. (*y*, *x*) ∈ *S*}

by (*induct S rule*: *finite.induct*) *auto*

lemma *finite-Collect-less-eq*:
Q ≤ *P* ⟹ *finite* (*Collect P*) ⟹ *finite* (*Collect Q*)
by (*metis* (*full-types*) *Ball-Collect infinite-iff-countable-subset rev-predicate1D*)

datatype *'a FSet-Lex-Wrapper* = *Wrapp* (*ex*: *'a fset*)

lemma *inj-FSet-Lex-Wrapper*: *inj Wrapp*
unfolding *inj-def* **by** *auto*

lemmas *ftrancl-map-both* = *inj-on-trancl-map-both*[*Transfer.transferred*]

instantiation *FSet-Lex-Wrapper* :: (*linorder*) *linorder*
begin

definition *less-eq-FSet-Lex-Wrapper* :: (*'a* :: *linorder*) *FSet-Lex-Wrapper* ⇒ *'a FSet-Lex-Wrapper* ⇒ *bool*
where *less-eq-FSet-Lex-Wrapper S T* =
(*let S'* = *sorted-list-of-fset* (*ex S*) *in*
let T' = *sorted-list-of-fset* (*ex T*) *in*
S' ≤ *T'*)

definition *less-FSet-Lex-Wrapper* :: *'a FSet-Lex-Wrapper* ⇒ *'a FSet-Lex-Wrapper* ⇒ *bool*
where *less-FSet-Lex-Wrapper S T* =
(*let S'* = *sorted-list-of-fset* (*ex S*) *in*
let T' = *sorted-list-of-fset* (*ex T*) *in*
S' < *T'*)

instance by (*intro-classes*)
(*auto simp*: *less-eq-FSet-Lex-Wrapper-def less-FSet-Lex-Wrapper-def ex-def FSet-Lex-Wrapper.expand dest*: *sorted-list-of-fset-id*)
end

end
theory *Ground-Ctxt*
imports *Ground-Terms*
begin

2.9.3 Ground context

datatype (*gfuns-ctxt*: *'f*) *gctxt* =
GHole (‹□$_G$›) | *GMore 'f 'f gterm list 'f gctxt 'f gterm list*
declare *gctxt.map-comp*[*simp*]

fun *gctxt-apply-term* :: *'f gctxt* ⇒ *'f gterm* ⇒ *'f gterm* (‹-⟨-⟩$_G$› [*1000*, *0*] *1000*)

where
  $\Box_G\langle s\rangle_G = s$ |
  $(GMore\ f\ ss1\ C\ ss2)\langle s\rangle_G = GFun\ f\ (ss1\ @\ C\langle s\rangle_G\ \#\ ss2)$

fun *hole-gpos* **where**
  hole-gpos $\Box_G = []$ |
  hole-gpos $(GMore\ f\ ss1\ C\ ss2) = length\ ss1\ \#\ hole\text{-}gpos\ C$

lemma *gctxt-eq* [*simp*]: $(C\langle s\rangle_G = C\langle t\rangle_G) = (s = t)$
  by (*induct C*) *auto*

fun *gctxt-compose* :: *'f gctxt* ⇒ *'f gctxt* ⇒ *'f gctxt* (**infixl** ‹$\circ_{Gc}$› *75*) **where**
  $\Box_G \circ_{Gc} D = D$ |
  $(GMore\ f\ ss1\ C\ ss2) \circ_{Gc} D = GMore\ f\ ss1\ (C \circ_{Gc} D)\ ss2$

fun *gctxt-at-pos* :: *'f gterm* ⇒ *pos* ⇒ *'f gctxt* **where**
  gctxt-at-pos t [] = $\Box_G$ |
  gctxt-at-pos $(GFun\ f\ ts)\ (i\ \#\ ps) =$
    $GMore\ f\ (take\ i\ ts)\ (gctxt\text{-}at\text{-}pos\ (ts\ !\ i)\ ps)\ (drop\ (Suc\ i)\ ts)$

interpretation *ctxt-monoid-mult*: *monoid-mult* $\Box_G\ (\circ_{Gc})$
proof
  fix *C D E* :: *'f gctxt*
  show $C \circ_{Gc} D \circ_{Gc} E = C \circ_{Gc} (D \circ_{Gc} E)$ **by** (*induct C*) *simp-all*
  show $\Box_G \circ_{Gc} C = C$ **by** *simp*
  show $C \circ_{Gc} \Box_G = C$ **by** (*induct C*) *simp-all*
qed

instantiation *gctxt* :: (*type*) *monoid-mult*
begin
  definition [*simp*]: *1* = $\Box_G$
  definition [*simp*]: $(*) = (\circ_{Gc})$
  instance by (*intro-classes*) (*simp-all add*: *ac-simps*)
end

lemma *ctxt-ctxt-compose* [*simp*]: $(C \circ_{Gc} D)\langle t\rangle_G = C\langle D\langle t\rangle_G\rangle_G$
  by (*induct C*) *simp-all*

lemmas *ctxt-ctxt* = *ctxt-ctxt-compose* [*symmetric*]

fun *ctxt-of-gctxt* **where**
  ctxt-of-gctxt $\Box_G = \Box$
| *ctxt-of-gctxt* $(GMore\ f\ ss\ C\ ts) = More\ f\ (map\ term\text{-}of\text{-}gterm\ ss)\ (ctxt\text{-}of\text{-}gctxt\ C)\ (map\ term\text{-}of\text{-}gterm\ ts)$

fun *gctxt-of-ctxt* **where**
  gctxt-of-ctxt $\Box = \Box_G$
| *gctxt-of-ctxt* $(More\ f\ ss\ C\ ts) = GMore\ f\ (map\ gterm\text{-}of\text{-}term\ ss)\ (gctxt\text{-}of\text{-}ctxt\ C)\ (map\ gterm\text{-}of\text{-}term\ ts)$

lemma *ground-ctxt-of-gctxt* [*simp*]:
ground-ctxt (*ctxt-of-gctxt s*)
by (*induct s*) *auto*

lemma *ground-ctxt-of-gctxt'* [*simp*]:
ctxt-of-gctxt C = *More f ss D ts* ⟹ *ground-ctxt* (*More f ss D ts*)
by (*induct C*) *auto*

lemma *ctxt-of-gctxt-inv* [*simp*]:
gctxt-of-ctxt (*ctxt-of-gctxt t*) = *t*
by (*induct t*) (*auto intro*!: *nth-equalityI*)

lemma *inj-ctxt-of-gctxt*: *inj-on ctxt-of-gctxt X*
by (*metis inj-on-def ctxt-of-gctxt-inv*)

lemma *gctxt-of-ctxt-inv* [*simp*]:
ground-ctxt C ⟹ *ctxt-of-gctxt* (*gctxt-of-ctxt C*) = *C*
by (*induct C*) (*auto 0 0 intro*!: *nth-equalityI*)

lemma *map-ctxt-of-gctxt* [*simp*]:
map-ctxt f g (*ctxt-of-gctxt C*) = *ctxt-of-gctxt* (*map-gctxt f C*)
by (*induct C*) *auto*

lemma *map-gctxt-of-ctxt* [*simp*]:
ground-ctxt C ⟹ *gctxt-of-ctxt* (*map-ctxt f g C*) = *map-gctxt f* (*gctxt-of-ctxt C*)
by (*induct C*) *auto*

lemma *map-gctxt-nempty* [*simp*]:
C ≠ □$_G$ ⟹ *map-gctxt f C* ≠ □$_G$
by (*cases C*) *auto*

lemma *gctxt-set-funs-ctxt*:
gfuns-ctxt C = *funs-ctxt* (*ctxt-of-gctxt C*)
using *gterm-set-gterm-funs-terms*
by (*induct C*) *fastforce*+

lemma *ctxt-set-funs-gctxt*:
assumes *ground-ctxt C*
shows *gfuns-ctxt* (*gctxt-of-ctxt C*) = *funs-ctxt C*
using *assms term-set-gterm-funs-terms*
by (*induct C*) *fastforce*+

lemma *vars-ctxt-of-gctxt* [*simp*]:
vars-ctxt (*ctxt-of-gctxt C*) = {}
by (*induct C*) *auto*

lemma *vars-ctxt-of-gctxt-subseteq* [*simp*]:
vars-ctxt (*ctxt-of-gctxt C*) ⊆ *Q* ⟷ *True*

by *auto*

lemma *term-of-gterm-ctxt-apply-ground* [*simp*]:
term-of-gterm $s = C\langle l \rangle \Longrightarrow$ *ground-ctxt* C
term-of-gterm $s = C\langle l \rangle \Longrightarrow$ *ground* l
by (*metis ground-ctxt-apply ground-term-of-gterm*)+

lemma *term-of-gterm-ctxt-subst-apply-ground* [*simp*]:
term-of-gterm $s = C\langle l \cdot \sigma \rangle \Longrightarrow x \in$ *vars-term* $l \Longrightarrow$ *ground* $(\sigma\ x)$
by (*meson ground-substD term-of-gterm-ctxt-apply-ground*(*2*))

lemma *gctxt-compose-HoleE*:
$C \circ_{Gc} D = \Box_G \Longrightarrow C = \Box_G$
$C \circ_{Gc} D = \Box_G \Longrightarrow D = \Box_G$
by (*cases* C; *cases* D, *auto*)+

— Relations between ground contexts and contexts

lemma *nempty-ground-ctxt-gctxt* [*simp*]:
$C \neq \Box \Longrightarrow$ *ground-ctxt* $C \Longrightarrow$ *gctxt-of-ctxt* $C \neq \Box_G$
by (*induct* C) *auto*

lemma *ctxt-of-gctxt-apply* [*simp*]:
gterm-of-term (*ctxt-of-gctxt* $C\langle$*term-of-gterm* $t\rangle = C\langle t \rangle_G$
by (*induct* C) (*auto simp*: *comp-def map-idI*)

lemma *ctxt-of-gctxt-apply-gterm*:
gterm-of-term (*ctxt-of-gctxt* $C\langle t\rangle = C\langle$*gterm-of-term* $t\rangle_G$
by (*induct* C) (*auto simp*: *comp-def map-idI*)

lemma *ground-gctxt-of-ctxt-apply-gterm*:
assumes *ground-ctxt* C
shows *term-of-gterm* (*gctxt-of-ctxt* $C\langle t\rangle_G = C\langle$*term-of-gterm* $t\rangle$ **using** *assms*
by (*induct* C) (*auto simp*: *comp-def map-idI*)

lemma *ground-gctxt-of-ctxt-apply* [*simp*]:
assumes *ground-ctxt* C *ground* t
shows *term-of-gterm* (*gctxt-of-ctxt* $C\langle$*gterm-of-term* $t\rangle_G = C\langle t\rangle$ **using** *assms*
by (*induct* C) (*auto simp*: *comp-def map-idI*)

lemma *term-of-gterm-ctxt-apply* [*simp*]:
term-of-gterm $s = C\langle l \rangle \Longrightarrow$ (*gctxt-of-ctxt* $C\langle$*gterm-of-term* $l\rangle_G = s$
by (*metis ctxt-of-gctxt-apply-gterm gctxt-of-ctxt-inv term-of-gterm-ctxt-apply-ground*(*1*) *term-of-gterm-inv*)

lemma *gctxt-apply-inj-term*: *inj* (*gctxt-apply-term* C)
by (*auto simp*: *inj-on-def*)

lemma *gctxt-apply-inj-on-term*: *inj-on* (*gctxt-apply-term C*) *S*
by (*auto simp*: *inj-on-def*)

lemma *ctxt-of-pos-gterm* [*simp*]:
$p \in gposs\ t \Longrightarrow$ *ctxt-at-pos* (*term-of-gterm t*) *p* = *ctxt-of-gctxt* (*gctxt-at-pos t p*)
by (*induct t arbitrary*: *p*) (*auto simp add*: *take-map drop-map*)

lemma *gctxt-of-gpos-gterm-gsubt-at-to-gterm* [*simp*]:
assumes $p \in gposs\ t$
shows $(gctxt\text{-}at\text{-}pos\ t\ p)\langle gsubt\text{-}at\ t\ p\rangle_G = t$ **using** *assms*
by (*induct t arbitrary*: *p*) (*auto simp*: *comp-def min-def nth-append-Cons intro*!: *nth-equalityI*)

The position of the hole in a context is uniquely determined

fun *ghole-pos* :: *'f gctxt* ⇒ *pos* **where**
$ghole\text{-}pos\ \Box_G = [] \mid$
ghole-pos (*GMore f ss D ts*) = *length ss* # *ghole-pos D*

lemma *ghole-pos-gctxt-at-pos* [*simp*]:
$p \in gposs\ t \Longrightarrow ghole\text{-}pos\ (gctxt\text{-}at\text{-}pos\ t\ p) = p$
by (*induct t arbitrary*: *p*) *auto*

lemma *ghole-pos-id-ctxt* [*simp*]:
$C\langle s\rangle_G = t \Longrightarrow gctxt\text{-}at\text{-}pos\ t\ (ghole\text{-}pos\ C) = C$
by (*induct C arbitrary*: *t*) *auto*

lemma *ghole-pos-in-apply*:
$ghole\text{-}pos\ C = p \Longrightarrow p \in gposs\ C\langle u\rangle_G$
by (*induct C arbitrary*: *p*) (*auto simp*: *nth-append*)

lemma *ground-hole-pos-to-ghole*:
ground-ctxt C ⟹ *ghole-pos* (*gctxt-of-ctxt C*) = *hole-pos C*
by (*induct C*) *auto*

lemma *gsubst-at-gctxt-at-eq-gtermD*:
assumes $s = t\ p \in gposs\ t$
shows $gsubt\text{-}at\ s\ p = gsubt\text{-}at\ t\ p \wedge gctxt\text{-}at\text{-}pos\ s\ p = gctxt\text{-}at\text{-}pos\ t\ p$ **using** *assms*
by *auto*

lemma *gsubst-at-gctxt-at-eq-gtermI*:
assumes $p \in gposs\ s\ p \in gposs\ t$
and $gsubt\text{-}at\ s\ p = gsubt\text{-}at\ t\ p$
and $gctxt\text{-}at\text{-}pos\ s\ p = gctxt\text{-}at\text{-}pos\ t\ p$
shows $s = t$ **using** *assms*
using *gctxt-of-gpos-gterm-gsubt-at-to-gterm* **by** *force*

lemma *gsubt-at-gctxt-apply-ghole* [*simp*]:

gsubt-at $C\langle u\rangle_G$ (*ghole-pos* C) = u
by (*induct* C) *auto*

lemma *gctxt-at-pos-gsubt-at-pos* [*simp*]:
$p \in$ *gposs* $t \Longrightarrow$ *gsubt-at* (*gctxt-at-pos* t p)$\langle u\rangle_G$ $p = u$
proof (*induct* p *arbitrary*: t)
case (*Cons* i p)
then show *?case* **using** *id-take-nth-drop*
by (*cases* t) (*auto simp*: *nth-append*)
qed *auto*

lemma *gfun-at-gctxt-at-pos-not-after*:
assumes $p \in$ *gposs* t $q \in$ *gposs* t $\neg$ ($p \leq_p q$)
shows *gfun-at* (*gctxt-at-pos* t p)$\langle v\rangle_G$ q = *gfun-at* t q **using** *assms*
proof (*induct* q *arbitrary*: p t)
case *Nil*
then show *?case*
by (*cases* p; *cases* t) *auto*
next
case (*Cons* i q)
from *Cons*(*4*) **obtain** j r **where** [*simp*]: $p = j \# r$ **by** (*cases* p) *auto*
from *Cons*(*4*) **have** $j = i \Longrightarrow \neg$ ($r \leq_p q$) **by** *auto*
from *this Cons*(*2*−) *Cons*(*1*)[*of* r *gargs* t ! j]
have $j = i \Longrightarrow$ *gfun-at* (*gctxt-at-pos* (*gargs* t ! j) r)$\langle v\rangle_G$ q = *gfun-at* (*gargs* t ! j) q
by (*cases* t) *auto*
then show *?case* **using** *Cons*(*2*, *3*)
by (*cases* t) (*auto simp*: *nth-append-Cons min-def*)
qed

lemma *gpos-less-eq-append* [*simp*]: $p \leq_p (p$ @ $q)$
unfolding *position-less-eq-def*
by *blast*

lemma *gposs-ConsE* [*elim*]:
assumes $i \# p \in$ *gposs* t
obtains f *ts* **where** t = *GFun* f *ts* *ts* $\neq$ [] $i <$ *length ts* $p \in$ *gposs* (*ts* ! i) **using** *assms*
by (*cases* t) *force*+

lemma *gposs-gctxt-at-pos-not-after*:
assumes $p \in$ *gposs* t $q \in$ *gposs* t $\neg$ ($p \leq_p q$)
shows $q \in$ *gposs* (*gctxt-at-pos* t p)$\langle v\rangle_G \longleftrightarrow q \in$ *gposs* t **using** *assms*
proof (*induct* q *arbitrary*: p t)
case *Nil* **then show** *?case*
by (*cases* p; *cases* t) *auto*
next
case (*Cons* i q)
from *Cons*(*4*) **obtain** j r **where** [*simp*]: $p = j \# r$ **by** (*cases* p) *auto*

from *Cons*(*4*) **have** $j = i \Longrightarrow \neg\ (r \leq_p q)$ **by** *auto*
from *this Cons*(*2*−) *Cons*(*1*)[*of r gargs t* ! *j*]
have $j = i \Longrightarrow q \in gposs\ (gctxt\text{-}at\text{-}pos\ (gargs\ t\ !\ j)\ r)\langle v\rangle_G \longleftrightarrow q \in gposs\ (gargs\ t\ !\ j)$
by (*cases t*) *auto*
then show *?case* **using** *Cons*(*2*, *3*)
by (*cases t*) (*auto simp*: *nth-append-Cons min-def*)
qed

lemma *gposs-gctxt-at-pos*:
$p \in gposs\ t \Longrightarrow gposs\ (gctxt\text{-}at\text{-}pos\ t\ p)\langle v\rangle_G = \{q.\ q \in gposs\ t \wedge \neg\ (p \leq_p q)\} \cup (@)\ p\ `\ gposs\ v$
proof (*induct p arbitrary*: *t*)
case (*Cons i p*)
show *?case* **using** *Cons*(*1*)[*of gargs t* ! *i*] *Cons*(*2*) *gposs-gctxt-at-pos-not-after*[*OF Cons*(*2*)]
by (*auto simp*: *min-def nth-append-Cons split*: *if-splits elim*!: *gposs-ConsE*)
qed *auto*

lemma *eq-gctxt-at-pos*:
assumes $p \in gposs\ s\ p \in gposs\ t$
and $\bigwedge q.\ \neg\ (p \leq_p q) \Longrightarrow q \in gposs\ s \longleftrightarrow q \in gposs\ t$
and $(\bigwedge q.\ q \in gposs\ s \Longrightarrow \neg\ (p \leq_p q) \Longrightarrow gfun\text{-}at\ s\ q = gfun\text{-}at\ t\ q)$
shows *gctxt-at-pos s p* = *gctxt-at-pos t p* **using** *assms*(*1*, *2*)
using *arg-cong*[**where** *?f* = *gctxt-of-ctxt*, *OF eq-ctxt-at-pos-by-poss*, *of* - *term-of-gterm s* :: (-, *unit*) *term*
term-of-gterm t :: (-, *unit*) *term* **for** *s t*, *unfolded poss-gposs-conv fun-at-gfun-at-conv ctxt-of-pos-gterm*,
OF assms]
by *simp*

Signature of a ground context

fun *funas-gctxt* :: *'f gctxt* ⇒ (*'f* × *nat*) *set* **where**
funas-gctxt GHole = {} |
funas-gctxt (*GMore f ss1 D ss2*) = {(*f*, *Suc* (*length* (*ss1* @ *ss2*)))}
∪ *funas-gctxt D* ∪ ⋃(*set* (*map funas-gterm* (*ss1* @ *ss2*)))

lemma *funas-gctxt-of-ctxt* [*simp*]:
ground-ctxt C ⟹ *funas-gctxt* (*gctxt-of-ctxt C*) = *funas-ctxt C*
by (*induct C*) (*auto simp*: *funas-gterm-gterm-of-term*)

lemma *funas-ctxt-of-gctxt-conv* [*simp*]:
funas-ctxt (*ctxt-of-gctxt C*) = *funas-gctxt C*
by (*induct C*) (*auto simp flip*: *funas-gterm-gterm-of-term*)

lemma *inj-gctxt-of-ctxt-on-ground*:
inj-on gctxt-of-ctxt (*Collect ground-ctxt*)
using *gctxt-of-ctxt-inv* **by** (*fastforce simp*: *inj-on-def*)

lemma *funas-gterm-ctxt-apply* [*simp*]:
funas-gterm $C\langle s\rangle_G$ = *funas-gctxt* $C \cup$ *funas-gterm* s
by (*induct* C) *auto*

lemma *funas-gctxt-compose* [*simp*]:
funas-gctxt ($C \circ_{Gc} D$) = *funas-gctxt* $C \cup$ *funas-gctxt* D
by (*induct* C *arbitrary*: D) *auto*

end
theory *Ground-Closure*
imports *Ground-Terms*
begin

2.9.4 Multihole context closure

Computing the multihole context closure of a given relation

inductive-set *gmctxt-cl* :: ('f × *nat*) *set* ⇒ 'f *gterm rel* ⇒ 'f *gterm rel* **for** $\mathcal{F}$ $\mathcal{R}$
where
base [*intro*]: $(s, t) \in \mathcal{R} \Longrightarrow (s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
| *step* [*intro*]: *length ss* = *length ts* ⟹ (∀ $i <$ *length ts*. (*ss* ! i, *ts* ! i) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$) ⟹ (f, *length ss*) ∈ $\mathcal{F}$ ⟹
($GFun\ f\ ss$, $GFun\ f\ ts$) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$

lemma *gmctxt-cl-idemp* [*simp*]:
gmctxt-cl $\mathcal{F}$ (*gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$) = *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
proof −
{fix s t **assume** $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ (*gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$)
then have $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
by (*induct*) (*auto intro*: *gmctxt-cl.step*)**}**
then show *?thesis* **by** *auto*
qed

lemma *gmctxt-cl-refl*:
funas-gterm $t \subseteq \mathcal{F} \Longrightarrow (t, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
by (*induct* t) (*auto simp*: *SUP-le-iff intro*!: *gmctxt-cl.step*)

lemma *gmctxt-cl-swap*:
gmctxt-cl $\mathcal{F}$ (*prod.swap* ‘ $\mathcal{R}$) = *prod.swap* ‘ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ (**is** *?Ls* = *?Rs*)
proof −
{fix s t **assume** $(s, t) \in$ *?Ls* **then have** $(s, t) \in$ *?Rs*
by *induct auto***}**
moreover
{fix s t **assume** $(s, t) \in$ *?Rs*
then have $(t, s) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ **by** *auto*
then have $(s, t) \in$ *?Ls* **by** *induct auto***}**
ultimately show *?thesis* **by** *auto*
qed

lemma *gmctxt-cl-mono-funas*:

assumes $\mathcal{F} \subseteq \mathcal{G}$ **shows** *gmctxt-cl* $\mathcal{F}$ $\mathcal{R} \subseteq$ *gmctxt-cl* $\mathcal{G}$ $\mathcal{R}$
proof −
{fix *s t* **assume** $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ **then have** $(s, t) \in$ *gmctxt-cl* $\mathcal{G}$ $\mathcal{R}$
by *induct* (*auto simp*: *subsetD*[*OF assms*])**}**
then show *?thesis* **by** *auto*
qed

lemma *gmctxt-cl-mono-rel*:
assumes $\mathcal{P} \subseteq \mathcal{R}$ **shows** *gmctxt-cl* $\mathcal{F}$ $\mathcal{P} \subseteq$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
proof −
{fix *s t* **assume** $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{P}$ **then have** $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ **using** *assms*
by *induct auto***}**
then show *?thesis* **by** *auto*
qed

definition *gcomp-rel* :: $('f \times nat)$ *set* $\Rightarrow$ *'f gterm rel* $\Rightarrow$ *'f gterm rel* $\Rightarrow$ *'f gterm rel* **where**
gcomp-rel $\mathcal{F}$ *R S* = (*R O gmctxt-cl* $\mathcal{F}$ *S*) $\cup$ (*gmctxt-cl* $\mathcal{F}$ *R O S*)

definition *gtrancl-rel* :: $('f \times nat)$ *set* $\Rightarrow$ *'f gterm rel* $\Rightarrow$ *'f gterm rel* **where**
gtrancl-rel $\mathcal{F}$ $\mathcal{R}$ = (*gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$)$^+$ *O* $\mathcal{R}$ *O* (*gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$)$^+$

lemma *gcomp-rel*:
gmctxt-cl $\mathcal{F}$ (*gcomp-rel* $\mathcal{F}$ $\mathcal{R}$ $\mathcal{S}$) = *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ *O gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$ (**is** *?Ls* = *?Rs*)
proof
{ fix *s u* **assume** $(s, u) \in$ *gmctxt-cl* $\mathcal{F}$ ($\mathcal{R}$ *O gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$ $\cup$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ *O* $\mathcal{S}$)
then have $\exists t.\ (s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R} \wedge (t, u) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$
proof (*induct*)
case (*step ss ts f*)
from *Ex-list-of-length-P*[*of* - λ *u i*. (*ss* ! *i*, *u*) $\in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R} \wedge$ (*u*, *ts* ! *i*) $\in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$]
obtain *us* **where** *l*: *length us* = *length ts* **and**
inv: $\forall$ *i* < *length ts*. (*ss* ! *i*, *us* ! *i*) $\in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R} \wedge$ (*us* ! *i*, *ts* ! *i*) $\in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$
using *step*(*2*, *3*) **by** *blast*
then show *?case* **using** *step*(*1*, *3*)
by (*intro exI*[*of* - *GFun f us*]) *auto*
qed *auto***}**
then show *?Ls* $\subseteq$ *?Rs* **unfolding** *gcomp-rel-def*
by *auto*
next
{fix *s t u* **assume** $(s, t) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ $(t, u) \in$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$
then have $(s, u) \in$ *gmctxt-cl* $\mathcal{F}$ ($\mathcal{R}$ *O gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$ $\cup$ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ *O* $\mathcal{S}$)
proof (*induct arbitrary*: *u rule*: *gmctxt-cl.induct*)
case (*step ss ts f*)
then show *?case*

proof (*cases* (*GFun f ts*, *u*) ∈ $\mathcal{S}$)
case *True*
then have (*GFun f ss*, *u*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ *O* $\mathcal{S}$ **using** *gmctxt-cl.step*[*OF step*(*1*) - *step*(*3*)] *step*(*2*)
by *auto*
then show *?thesis* **by** *auto*
next
case *False*
then obtain *us* **where** *u*[*simp*]: *u* = *GFun f us* **and** *l*: *length ts* = *length us*
using *step*(*4*) **by** (*cases u*) (*auto elim*: *gmctxt-cl.cases*)
have *i* < *length us* ⟹
(*ss* ! *i*, *us* ! *i*) ∈ *gmctxt-cl* $\mathcal{F}$ ($\mathcal{R}$ *O gmctxt-cl* $\mathcal{F}$ $\mathcal{S}$ ∪ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ *O* $\mathcal{S}$)
for *i*
using *step*(*1*, *2*, *4*) *False* **by** (*auto elim*: *gmctxt-cl.cases*)
then show *?thesis* **using** *l step*(*1*, *3*)
by *auto*
qed
qed *auto*}
then show *?Rs* ⊆ *?Ls*
by (*auto simp*: *gcomp-rel-def*)
qed

2.9.5 Signature closed property

definition *all-ctxt-closed* :: ('*f* × *nat*) *set* ⇒ '*f gterm rel* ⇒ *bool* **where**
all-ctxt-closed F r ⟷ (∀ *f ts ss*. (*f*, *length ss*) ∈ *F* ⟶ *length ss* = *length ts* ⟶
(∀ *i*. *i* < *length ts* ⟶ (*ss* ! *i*, *ts* ! *i*) ∈ *r*) ⟶
(*GFun f ss*, *GFun f ts*) ∈ *r*)

lemma *all-ctxt-closedI*:
assumes ⋀ *f ss ts*. (*f*, *length ss*) ∈ $\mathcal{F}$ ⟹ *length ss* = *length ts* ⟹
(∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *r*) ⟹ (*GFun f ss*, *GFun f ts*) ∈ *r*
shows *all-ctxt-closed* $\mathcal{F}$ *r* **using** *assms*
unfolding *all-ctxt-closed-def* **by** *auto*

lemma *all-ctxt-closedD*:
all-ctxt-closed F r ⟹ (*f*, *length ss*) ∈ *F* ⟹ *length ss* = *length ts* ⟹
(∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *r*) ⟹ (*GFun f ss*, *GFun f ts*) ∈ *r*
by (*auto simp*: *all-ctxt-closed-def*)

lemma *all-ctxt-closed-refl-on*:
assumes *all-ctxt-closed* $\mathcal{F}$ *r s* ∈ $\mathcal{T}_G$ $\mathcal{F}$
shows (*s*, *s*) ∈ *r* **using** *assms*(*2*)
by (*induct*) (*auto simp*: *all-ctxt-closedD*[*OF assms*(*1*)])

lemma *gmctxt-cl-is-all-ctxt-closed* [*simp*]:
all-ctxt-closed $\mathcal{F}$ (*gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$)
unfolding *all-ctxt-closed-def*

by *auto*

lemma *all-ctxt-closed-gmctxt-cl-idem* [*simp*]:
assumes *all-ctxt-closed* $\mathcal{F}$ $\mathcal{R}$
shows *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ = $\mathcal{R}$
proof −
{fix *s t* **assume** (*s*, *t*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$ **then have** (*s*, *t*) ∈ $\mathcal{R}$
proof (*induct*)
case (*step ss ts f*)
show *?case* **using** *step*(*2*) *all-ctxt-closedD*[*OF assms step*(*3*, *1*)]
by *auto*
qed *auto***}**
then show *?thesis* **by** *auto*
qed

2.9.6 Transitive closure preserves *all-ctxt-closed*

induction scheme for transitive closures of lists

inductive-set *trancl-list* **for** $\mathcal{R}$ **where**
base[*intro*, *Pure.intro*] : *length xs* = *length ys* ⟹
(∀ *i* < *length ys*. (*xs* ! *i*, *ys* ! *i*) ∈ $\mathcal{R}$) ⟹ (*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$
| *list-trancl* [*Pure.intro*]: (*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$ ⟹ *i* < *length ys* ⟹ (*ys* ! *i*, *z*) ∈ $\mathcal{R}$ ⟹
(*xs*, *ys*[*i* := *z*]) ∈ *trancl-list* $\mathcal{R}$

lemma *trancl-list-appendI* [*simp*, *intro*]:
(*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$ ⟹ (*x*, *y*) ∈ $\mathcal{R}$ ⟹ (*x* # *xs*, *y* # *ys*) ∈ *trancl-list* $\mathcal{R}$
proof (*induct rule*: *trancl-list.induct*)
case (*base xs ys*)
then show *?case* **using** *less-Suc-eq-0-disj*
by (*intro trancl-list.base*) *auto*
next
case (*list-trancl xs ys i z*)
from *list-trancl*(*3*) **have** ∗: *y* # *ys*[*i* := *z*] = (*y* # *ys*)[*Suc i* := *z*] **by** *auto*
show *?case* **using** *list-trancl* **unfolding** ∗
by (*intro trancl-list.list-trancl*) *auto*
qed

lemma *trancl-list-append-tranclI* [*intro*]:
(*x*, *y*) ∈ $\mathcal{R}^+$ ⟹ (*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$ ⟹ (*x* # *xs*, *y* # *ys*) ∈ *trancl-list* $\mathcal{R}$
proof (*induct rule*: *trancl.induct*)
case (*trancl-into-trancl a b c*)
then have (*a* # *xs*, *b* # *ys*) ∈ *trancl-list* $\mathcal{R}$ **by** *auto*
from *trancl-list.list-trancl*[*OF this*, *of 0 c*]
show *?case* **using** *trancl-into-trancl*(*3*)
by *auto*
qed *auto*

lemma *trancl-list-conv*:

```
  (xs, ys) ∈ trancl-list R ⟷ length xs = length ys ∧ (∀ i < length ys. (xs ! i, ys
! i) ∈ R⁺) (is ?Ls ⟷ ?Rs)
proof
  assume ?Ls then show ?Rs
  proof (induct)
    case (list-trancl xs ys i z)
    then show ?case
      by auto (metis nth-list-update trancl.trancl-into-trancl)
  qed auto
next
  assume ?Rs then show ?Ls
  proof (induct ys arbitrary: xs)
    case Nil
    then show ?case by (cases xs) auto
  next
    case (Cons y ys)
    from Cons(2) obtain x xs' where *: xs = x # xs' and
      inv: (x, y) ∈ R⁺
      by (cases xs) auto
    show ?case using Cons(1)[of tl xs] Cons(2) unfolding *
      by (intro trancl-list-append-tranclI[OF inv]) force
  qed
qed

lemma trancl-list-induct [consumes 2, case-names base step]:
  assumes length ss = length ts ∀ i < length ts. (ss ! i, ts ! i) ∈ R⁺
    and ⋀xs ys. length xs = length ys ⟹ ∀ i < length ys. (xs ! i, ys ! i) ∈ R ⟹
P xs ys
    and ⋀xs ys i z. length xs = length ys ⟹ ∀ i < length ys. (xs ! i, ys ! i) ∈ R⁺
⟹ P xs ys
      ⟹ i < length ys ⟹ (ys ! i, z) ∈ R ⟹ P xs (ys[i := z])
  shows P ss ts using assms
  by (intro trancl-list.induct[of ss ts R P]) (auto simp: trancl-list-conv)

lemma trancl-list-all-step-induct [consumes 2, case-names base step]:
  assumes length ss = length ts ∀ i < length ts. (ss ! i, ts ! i) ∈ R⁺
    and base: ⋀xs ys. length xs = length ys ⟹ ∀ i < length ys. (xs ! i, ys ! i) ∈
R ⟹ P xs ys
    and steps: ⋀xs ys zs. length xs = length ys ⟹ length ys = length zs ⟹
      ∀ i < length zs. (xs ! i, ys ! i) ∈ R⁺ ⟹ ∀ i < length zs. (ys ! i, zs ! i) ∈
R ∨ ys ! i = zs ! i ⟹
      P xs ys ⟹ P xs zs
  shows P ss ts using assms(1, 2)
proof (induct rule: trancl-list-induct)
case (step xs ys i z)
  then show ?case
    by (intro steps[of xs ys ys[i := z]])
      (auto simp: nth-list-update)
qed (auto simp: base)
```

lemma *all-ctxt-closed-trancl*:
assumes *all-ctxt-closed* $\mathcal{F}$ $\mathcal{R}$ $\mathcal{R} \subseteq \mathcal{T}_G\ \mathcal{F} \times \mathcal{T}_G\ \mathcal{F}$
shows *all-ctxt-closed* $\mathcal{F}$ $(\mathcal{R}^+)$
proof −
{fix *f ss ts* **assume** *sig*: $(f, \mathit{length}\ ss) \in \mathcal{F}$ **and**
steps: *length ss* = *length ts* $\forall\, i{<}\mathit{length}\ ts.\ (ss\ !\ i, ts\ !\ i) \in \mathcal{R}^+$
have $(\mathit{GFun}\ f\ ss, \mathit{GFun}\ f\ ts) \in \mathcal{R}^+$ **using** *steps sig*
proof (*induct rule*: *trancl-list-induct*)
case (*base ss ts*)
then show *?case* **using** *all-ctxt-closedD*[*OF assms*(*1*) *base*(*3*, *1*, *2*)]
by *auto*
next
case (*step ss ts i t'*)
from *step*(*2*) **have** $j < \mathit{length}\ ts \Longrightarrow ts\ !\ j \in \mathcal{T}_G\ \mathcal{F}$ **for** *j* **using** *assms*(*2*)
by (*metis* (*no-types*, *lifting*) *SigmaD2 subset-iff trancl.simps*)
from *this*[*THEN all-ctxt-closed-refl-on*[*OF assms*(*1*)]]
have $(\mathit{GFun}\ f\ ts, \mathit{GFun}\ f\ (ts[i := t'])) \in \mathcal{R}$ **using** *step*(*1*, *4*−)
by (*intro all-ctxt-closedD*[*OF assms*(*1*)]) (*auto simp*: *nth-list-update*)
then show *?case* **using** *step*(*3*, *6*)
by *auto*
qed}
then show *?thesis* **by** (*intro all-ctxt-closedI*)
qed

end
theory *Horn-Inference*
imports *Main*
begin

datatype *'a horn* = *horn 'a list 'a* (**infix** $\langle\rightarrow_h\rangle$ *55*)

locale *horn* =
fixes $\mathcal{H}$ :: *'a horn set*
begin

inductive-set *saturate* :: *'a set* **where**
infer: *as* $\rightarrow_h$ *a* $\in \mathcal{H} \Longrightarrow (\bigwedge x.\ x \in \mathit{set}\ as \Longrightarrow x \in \mathit{saturate}) \Longrightarrow a \in \mathit{saturate}$

definition *infer0* **where**
infer0 = $\{a.\ [] \rightarrow_h a \in \mathcal{H}\}$

definition *infer1* **where**
infer1 x B = $\{a\ |as\ a.\ as \rightarrow_h a \in \mathcal{H} \wedge x \in \mathit{set}\ as \wedge \mathit{set}\ as \subseteq B \cup \{x\}\}$

inductive *step* :: *'a set* × *'a set* ⇒ *'a set* × *'a set* ⇒ *bool* (**infix** $\langle\vdash\rangle$ *50*) **where**
delete: $x \in B \Longrightarrow (\mathit{insert}\ x\ G, B) \vdash (G, B)$
| *propagate*: $(\mathit{insert}\ x\ G, B) \vdash (G \cup \mathit{infer1}\ x\ B, \mathit{insert}\ x\ B)$
| *refl*: $(G, B) \vdash (G, B)$

| *trans*: $(G,\ B) \vdash (G',\ B') \Longrightarrow (G',\ B') \vdash (G'',\ B'') \Longrightarrow (G,\ B) \vdash (G'',\ B'')$

lemma *step-mono*:
$(G,\ B) \vdash (G',\ B') \Longrightarrow (H \cup G,\ B) \vdash (H \cup G',\ B')$
by (*induction* $(G,\ B)$ $(G',\ B')$ *arbitrary*: $G\ B\ G'\ B'$ *rule*: *step.induct*)
(*auto intro*: *step.intros simp*: *Un-assoc*[*symmetric*])

fun *invariant* **where**
invariant $(G,\ B) \longleftrightarrow G \subseteq saturate \wedge B \subseteq saturate \wedge (\forall a\ as.\ as \rightarrow_h a \in \mathcal{H} \wedge set\ as \subseteq B \longrightarrow a \in G \cup B)$

lemma *inv-start*:
shows *invariant* (*infer0*, {})
by (*auto simp*: *infer0-def invariant.simps intro*: *infer*)

lemma *inv-step*:
assumes *invariant* $(G,\ B)$ $(G,\ B) \vdash (G',\ B')$
shows *invariant* $(G',\ B')$
using *assms*(*2*,*1*)
proof (*induction* $(G,\ B)$ $(G',\ B')$ *arbitrary*: $G\ B\ G'\ B'$ *rule*: *step.induct*)
case (*propagate x G B*)
let *?G'* $= G \cup$ *local.infer1 x B* **and** *?B'* $=$ *insert x B*
have *?G'* $\subseteq$ *saturate ?B'* $\subseteq$ *saturate*
using *assms*(*1*) *propagate* **by** (*auto 0 3 simp*: *infer1-def intro*: *saturate.infer*)
moreover have *as* $\rightarrow_h a \in \mathcal{H} \Longrightarrow$ *set as* $\subseteq$ *?B'* $\Longrightarrow a \in$ *?G'* $\cup$ *?B'* **for** *a as*
using *assms*(*1*) *propagate* **by** (*fastforce simp*: *infer1-def*)
ultimately show *?case* **by** *auto*
qed *auto*

lemma *inv-end*:
assumes *invariant* ({}, *B*)
shows $B =$ *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
case (*lr x*) **then show** *?case* **using** *assms* **by** *auto*
next
case (*rl x*) **then show** *?case* **using** *assms*
by (*induct x rule*: *saturate.induct*) *fastforce*
qed

lemma *step-sound*:
(*infer0*, {}) $\vdash$ ({}, *B*) $\Longrightarrow B =$ *saturate*
by (*metis inv-start inv-step inv-end*)

end

lemma *horn-infer0-union*:
horn.infer0 $(\mathcal{H}_1 \cup \mathcal{H}_2)$ = *horn.infer0* $\mathcal{H}_1 \cup$ *horn.infer0* $\mathcal{H}_2$
by (*auto simp*: *horn.infer0-def*)

lemma *horn-infer1-union*:
horn.infer1 $(\mathcal{H}_1 \cup \mathcal{H}_2)$ *x B* = *horn.infer1* $\mathcal{H}_1$ *x B* ∪ *horn.infer1* $\mathcal{H}_2$ *x B*
by (*auto simp*: *horn.infer1-def*)

end
theory *Horn-List*
imports *Horn-Inference*
begin

locale *horn-list-impl* = *horn* +
fixes *infer0-impl* :: *'a list* **and** *infer1-impl* :: *'a* ⇒ *'a list* ⇒ *'a list*
begin

lemma *saturate-fold-simp* [*simp*]:
fold (λ*xa*. *case-option None* (*f xa*)) *xs None* = *None*
by (*induct xs*) *auto*

lemma *saturate-fold-mono* [*partial-function-mono*]:
option.mono-body (λ*f*. *fold* (λ*x*. *case-option None* (λ*y*. *f* (*x*, *y*))) *xs b*)
unfolding *monotone-def fun-ord-def flat-ord-def*
proof (*intro allI impI*, *induct xs arbitrary*: *b*)
case (*Cons a xs*)
show *?case*
using *Cons*(*1*)[*OF Cons*(*2*), *of x* (*a*, *the b*)] *Cons*(*2*)[*rule-format*, *of* (*a*, *the b*)]
by (*cases b*) *auto*
qed *auto*

partial-function (*option*) *saturate-rec* :: *'a* ⇒ *'a list* ⇒ (*'a list*) *option* **where**
saturate-rec x bs = (*if x* ∈ *set bs then Some bs else*
fold (λ*x*. *case-option None* (*saturate-rec x*)) (*infer1-impl x bs*) (*Some* (*x* # *bs*)))

definition *saturate-impl* **where**
saturate-impl = *fold* (λ*x*. *case-option None* (*saturate-rec x*)) *infer0-impl* (*Some* [])

end

locale *horn-list* = *horn-list-impl* +
assumes *infer0*: *infer0* = *set infer0-impl*
and *infer1*: ⋀*x bs*. *infer1 x* (*set bs*) = *set* (*infer1-impl x bs*)
begin

lemma *saturate-rec-sound*:
saturate-rec x bs = *Some bs'* ⟹ ({*x*}, *set bs*) ⊢ ({}, *set bs'*)
proof (*induct arbitrary*: *x bs bs' rule*: *saturate-rec.fixp-induct*)
case *1* **show** *?case* **using** *option-admissible*[*of* λ(*x*, *y*) *z*. *- x y z*]
by *fastforce*
next

```
  case (3 rec)
  have [dest!]: (set xs, set ys) ⊢ ({}, set bs')
    if fold (λx a. case a of None ⇒ None | Some a ⇒ rec x a) xs (Some ys) =
Some bs'
    for xs ys using that
  proof (induct xs arbitrary: ys)
    case (Cons a xs)
    show ?case using trans[OF step-mono[OF 3(1)], of a ys - set xs {} set bs']
Cons
      by (cases rec a ys) auto
  qed (auto intro: refl)
  show ?case using propagate[of x {} set bs, unfolded infer1 Un-empty-left] 3(2)
    by (auto split: if-splits intro: trans delete)
qed auto

lemma saturate-impl-sound:
  assumes saturate-impl = Some B'
  shows set B' = saturate
proof −
  have (set xs, set ys) ⊢ ({}, set bs')
    if fold (λx a. case a of None ⇒ None | Some a ⇒ saturate-rec x a) xs (Some
ys) = Some bs'
    for xs ys bs' using that
  proof (induct xs arbitrary: ys)
    case (Cons a xs)
    show ?case
      using trans[OF step-mono[OF saturate-rec-sound], of a ys - set xs {} set bs']
Cons
      by (cases saturate-rec a ys) auto
  qed (auto intro: refl)
  from this[of infer0-impl [] B'] assms step-sound show ?thesis
    by (auto simp: saturate-impl-def infer0)
qed

lemma saturate-impl-complete:
  assumes finite saturate
  shows saturate-impl ≠ None
proof −
  have ∗: fold (λx. case-option None (saturate-rec x)) ds (Some bs) ≠ None
    if set bs ⊆ saturate set ds ⊆ saturate for bs ds
    using that
  proof (induct card (saturate − set bs) arbitrary: bs ds rule: less-induct)
    case less
    show ?case using less(3)
    proof (induct ds)
      case (Cons d ds)
      have infer1 d (set bs) ⊆ saturate using less(2) Cons(2)
        unfolding infer1-def by (auto intro: saturate.infer)
      moreover have card (saturate − set (d # bs)) < card (saturate − set bs) if
```

```
d ∉ set bs
        using Cons(2) assms that
      by (metis (no-types, lifting) DiffI card-Diff1-less-iff card-Diff-insert card-Diff-singleton-if
finite-Diff list.set-intros(1) list.simps(15) subsetD)
     ultimately show ?case using less(1)[of d # bs infer1-impl d bs @ ds] less(2)
Cons assms
        unfolding fold.simps comp-def option.simps
        by (subst saturate-rec.simps) (auto split: if-splits simp: infer1)
    qed simp
  qed
  show ?thesis using ∗[of [] infer0-impl] inv-start by (simp add: saturate-impl-def
infer0)
qed

end

lemmas [code] = horn-list-impl.saturate-rec.simps horn-list-impl.saturate-impl-def

end
theory Horn-Fset
  imports Horn-Inference FSet-Utils
begin

locale horn-fset-impl = horn +
  fixes infer0-impl :: 'a list and infer1-impl :: 'a ⇒ 'a fset ⇒ 'a list
begin

lemma saturate-fold-simp [simp]:
  fold (λxa. case-option None (f xa)) xs None = None
  by (induct xs) auto

lemma saturate-fold-mono [partial-function-mono]:
  option.mono-body (λf. fold (λx. case-option None (λy. f (x, y))) xs b)
  unfolding monotone-def fun-ord-def flat-ord-def
proof (intro allI impI, induct xs arbitrary: b)
  case (Cons a xs)
  show ?case
    using Cons(1)[OF Cons(2), of x (a, the b)] Cons(2)[rule-format, of (a, the b)]
    by (cases b) auto
qed auto

partial-function (option) saturate-rec :: 'a ⇒ 'a fset ⇒ ('a fset) option where
  saturate-rec x bs = (if x |∈| bs then Some bs else
     fold (λx. case-option None (saturate-rec x)) (infer1-impl x bs) (Some (finsert
x bs)))

definition saturate-impl where
  saturate-impl = fold (λx. case-option None (saturate-rec x)) infer0-impl (Some
{||})
```

end

locale *horn-fset* = *horn-fset-impl* +
  assumes *infer0*: *infer0* = *set infer0-impl*
    and *infer1*: $\bigwedge x\ bs.$ *infer1 x* (*fset bs*) = *set* (*infer1-impl x bs*)
begin

lemma *saturate-rec-sound*:
  saturate-rec x bs = *Some bs'* $\Longrightarrow$ ({*x*}, *fset bs*) $\vdash$ ({}, *fset bs'*)
proof (*induct arbitrary*: *x bs bs' rule*: *saturate-rec.fixp-induct*)
  case *1* **show** *?case* **using** *option-admissible*[*of* $\lambda(x,\ y)\ z.$ - *x y z*]
    by *fastforce*
next
  case (*3 rec*)
  have [*dest*!]: (*set xs*, *fset ys*) $\vdash$ ({}, *fset bs'*)
    if *fold* ($\lambda x\ a.$ *case a of None* $\Rightarrow$ *None* | *Some a* $\Rightarrow$ *rec x a*) *xs* (*Some ys*) = *Some bs'*
    for *xs ys* **using** *that*
  proof (*induct xs arbitrary*: *ys*)
    case (*Cons a xs*)
    show *?case* **using** *trans*[*OF step-mono*[*OF 3*(*1*)], *of a ys* - *set xs* {} *fset bs'*] *Cons*
      by (*cases rec a ys*) *auto*
  qed (*auto intro*: *refl*)
  show *?case* **using** *propagate*[*of x* {} *fset bs*, *unfolded infer1 Un-empty-left*] *3*(*2*)
    by (*auto simp*: *delete split*: *if-splits intro*: *trans delete*)
qed *auto*

lemma *saturate-impl-sound*:
  assumes *saturate-impl* = *Some B'*
  shows *fset B'* = *saturate*
proof −
  have (*set xs*, *fset ys*) $\vdash$ ({}, *fset bs'*)
    if *fold* ($\lambda x\ a.$ *case a of None* $\Rightarrow$ *None* | *Some a* $\Rightarrow$ *saturate-rec x a*) *xs* (*Some ys*) = *Some bs'*
    for *xs ys bs'* **using** *that*
  proof (*induct xs arbitrary*: *ys*)
    case (*Cons a xs*)
    show *?case*
      using *trans*[*OF step-mono*[*OF saturate-rec-sound*], *of a ys* - *set xs* {} *fset bs'*] *Cons*
      by (*cases saturate-rec a ys*) *auto*
  qed (*auto intro*: *refl*)
  from *this*[*of infer0-impl* {||} *B'*] *assms step-sound* **show** *?thesis*
    by (*auto simp*: *saturate-impl-def infer0*)
qed

lemma *saturate-impl-complete*:

assumes *finite saturate*
shows *saturate-impl* ≠ *None*
proof −
have ∗: *fold* (λ*x*. *case-option None* (*saturate-rec x*)) *ds* (*Some bs*) ≠ *None*
if *fset bs* ⊆ *saturate set ds* ⊆ *saturate* **for** *bs ds*
using *that*
proof (*induct card* (*saturate* − *fset bs*) *arbitrary*: *bs ds rule*: *less-induct*)
case *less*
show *?case* **using** *less*(*3*)
proof (*induct ds*)
case (*Cons d ds*)
have *infer1 d* (*fset bs*) ⊆ *saturate* **using** *less*(*2*) *Cons*(*2*)
unfolding *infer1-def* **by** (*auto intro*: *saturate.infer*)
moreover have *card* (*saturate* − *fset* (*finsert d bs*)) < *card* (*saturate* − *fset bs*) **if** *d* ∉ *fset bs*
using *Cons*(*2*) *assms that*
by (*metis DiffI Diff-insert card-Diff1-less finite-Diff finsert.rep-eq in-mono insertCI list.simps*(*15*))
ultimately show *?case* **using** *less*(*1*)[*of finsert d bs infer1-impl d bs* @ *ds*] *less*(*2*) *Cons assms*
unfolding *fold.simps comp-def option.simps*
apply (*subst saturate-rec.simps*)
apply (*auto simp flip*: *saturate-rec.simps split*!: *if-splits simp*: *infer1*)
apply (*simp add*: *saturate-rec.simps*)
done
qed *simp*
qed
show *?thesis* **using** ∗[*of* {||} *infer0-impl*] *inv-start* **by** (*simp add*: *saturate-impl-def infer0*)
qed

end

lemmas [*code*] = *horn-fset-impl.saturate-rec.simps horn-fset-impl.saturate-impl-def*

end

3 Tree automaton

theory *Tree-Automata*
imports *FSet-Utils*
HOL−Library.Product-Lexorder
HOL−Library.Option-ord
begin

3.1 Tree automaton definition and functionality

datatype ('*q*, '*f*) *ta-rule* = *TA-rule* (*r-root*: '*f*) (*r-lhs-states*: '*q list*) (*r-rhs*: '*q*) (‹- - → -› [*51*, *51*, *51*] *52*)

datatype $('q, 'f)$ *ta* = *TA* (*rules*: $('q, 'f)$ *ta-rule fset*) (*eps*: $('q \times 'q)$ *fset*)

In many application we are interested in specific subset of all terms. If these can be captured by a tree automaton (identified by a state) then we say the set is regular. This gives the motivation for the following definition

datatype $('q, 'f)$ *reg* = *Reg* (*fin*: $'q$ *fset*) (*ta*: $('q, 'f)$ *ta*)

The state set induced by a tree automaton is implicit in our representation. We compute it based on the rules and epsilon transitions of a given tree automaton

abbreviation *rule-arg-states* **where** *rule-arg-states* $\Delta \equiv |\bigcup|$ ((*fset-of-list* $\circ$ *r-lhs-states*) $|`|$ Δ)
abbreviation *rule-target-states* **where** *rule-target-states* $\Delta \equiv$ (*r-rhs* $|`|$ Δ)
definition *rule-states* **where** *rule-states* $\Delta \equiv$ *rule-arg-states* Δ $|\cup|$ *rule-target-states* Δ

definition *eps-states* **where** *eps-states* $\Delta_\varepsilon \equiv$ (*fst* $|`|$ $\Delta_\varepsilon$) $|\cup|$ (*snd* $|`|$ Δ_ε)
definition $\mathcal{Q}$ $\mathcal{A}$ = *rule-states* (*rules* $\mathcal{A}$) $|\cup|$ *eps-states* (*eps* $\mathcal{A}$)
abbreviation $\mathcal{Q}_r$ $\mathcal{A} \equiv \mathcal{Q}$ (*ta* $\mathcal{A}$)

definition *ta-rhs-states* :: $('q, 'f)$ *ta* $\Rightarrow$ $'q$ *fset* **where**
ta-rhs-states $\mathcal{A} \equiv \{|\ q \mid p\ q.\ (p\ |\in|$ *rule-target-states* (*rules* $\mathcal{A}$)) $\wedge$ ($p = q \vee (p,$ $q)\ |\in|$ (*eps* $\mathcal{A}$)$|^+|$)$|\}$

definition *ta-sig* $\mathcal{A} = (\lambda\ r.\ (r\text{-}root\ r,\ length\ (r\text{-}lhs\text{-}states\ r)))\ |`|\ (rules\ \mathcal{A})$

3.1.1 Rechability of a term induced by a tree automaton

fun *ta-der* :: $('q, 'f)$ *ta* $\Rightarrow$ $('f, 'q)$ *term* $\Rightarrow$ $'q$ *fset* **where**
ta-der $\mathcal{A}$ (*Var* q) = $\{|q' \mid q'.\ q = q' \vee (q, q')\ |\in|$ (*eps* $\mathcal{A}$)$|^+|\ |\}$
$|$ *ta-der* $\mathcal{A}$ (*Fun* f *ts*) = $\{|q' \mid q'\ q\ qs.$
TA-rule f *qs* $q\ |\in|$ (*rules* $\mathcal{A}$) $\wedge$ ($q = q' \vee (q, q')\ |\in|$ (*eps* $\mathcal{A}$)$|^+|$) $\wedge$ *length qs* = *length ts* $\wedge$
($\forall\ i <$ *length ts*. *qs* ! $i\ |\in|$ *ta-der* $\mathcal{A}$ (*ts* ! i))$|\}$

fun *ta-der′* :: $('q,'f)$ *ta* $\Rightarrow$ $('f,'q)$ *term* $\Rightarrow$ $('f,'q)$ *term fset* **where**
ta-der′ $\mathcal{A}$ (*Var* p) = $\{|$*Var* $q \mid q.\ p = q \vee\ (p, q)\ |\in|$ (*eps* $\mathcal{A}$)$|^+|\ |\}$
$|$ *ta-der′* $\mathcal{A}$ (*Fun* f *ts*) = $\{|$*Var* $q \mid q.\ q\ |\in|$ *ta-der* $\mathcal{A}$ (*Fun* f *ts*)$|\}$ $|\cup|$
$\{|$*Fun* f *ss* $\mid$ *ss*. *length ss* = *length ts* $\wedge$
($\forall\ i <$ *length ts*. *ss* ! $i\ |\in|$ *ta-der′* $\mathcal{A}$ (*ts* ! i))$|\}$

Sometimes it is useful to analyse a concrete computation done by a tree automaton. To do this we introduce the notion of run which keeps track which states are computed in each subterm to reach a certain state.

abbreviation *ex-rule-state* $\equiv$ *fst* $\circ$ *groot-sym*
abbreviation *ex-comp-state* $\equiv$ *snd* $\circ$ *groot-sym*

inductive *run* **for** $\mathcal{A}$ **where**

step: *length qs* = *length ts* $\Longrightarrow$ ($\forall$ *i* < *length ts*. *run* $\mathcal{A}$ (*qs* ! *i*) (*ts* ! *i*)) $\Longrightarrow$
TA-rule f (*map ex-comp-state qs*) *q* |∈| (*rules* $\mathcal{A}$) $\Longrightarrow$ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|) $\Longrightarrow$
run $\mathcal{A}$ (*GFun* (*q*, *q'*) *qs*) (*GFun f ts*)

3.1.2 Language acceptance

definition *ta-lang* :: '*q fset* ⇒ ('*q*, '*f*) *ta* ⇒ ('*f*, '*v*) *terms* **where**
[*code del*]: *ta-lang Q* $\mathcal{A}$ = {*adapt-vars t* | *t*. *ground t* ∧ *Q* |∩| *ta-der* $\mathcal{A}$ *t* ≠ {||}}

definition *gta-der* **where**
gta-der $\mathcal{A}$ *t* = *ta-der* $\mathcal{A}$ (*term-of-gterm t*)

definition *gta-lang* **where**
gta-lang Q $\mathcal{A}$ = {*t*. *Q* |∩| *gta-der* $\mathcal{A}$ *t* ≠ {||}}

definition $\mathcal{L}$ **where**
$\mathcal{L}$ $\mathcal{A}$ = *gta-lang* (*fin* $\mathcal{A}$) (*ta* $\mathcal{A}$)

definition *reg-Restr-*Q_f **where**
reg-Restr-Q_f *R* = *Reg* (*fin R* |∩| $\mathcal{Q}_r$ *R*) (*ta R*)

3.1.3 Trimming

definition *ta-restrict* **where**
ta-restrict $\mathcal{A}$ *Q* = *TA* {| *TA-rule f qs q*| *f qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ *fset-of-list qs* |⊆| *Q* ∧ *q* |∈| *Q* |} (*fRestr* (*eps* $\mathcal{A}$) *Q*)

definition *ta-reachable* :: ('*q*, '*f*) *ta* ⇒ '*q fset* **where**
ta-reachable $\mathcal{A}$ = {|*q*| *q*. ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t* |}

definition *ta-productive* :: '*q fset* ⇒ ('*q*,'*f*) *ta* ⇒ '*q fset* **where**
ta-productive P $\mathcal{A}$ ≡ {|*q*| *q q' C*. *q'* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*Var q*⟩) ∧ *q'* |∈| *P* |}

An automaton is trim if all its states are reachable and productive.

definition *ta-is-trim* :: '*q fset* ⇒ ('*q*, '*f*) *ta* ⇒ *bool* **where**
ta-is-trim P $\mathcal{A}$ ≡ ∀ *q*. *q* |∈| $\mathcal{Q}$ $\mathcal{A}$ ⟶ *q* |∈| *ta-reachable* $\mathcal{A}$ ∧ *q* |∈| *ta-productive P* $\mathcal{A}$

definition *reg-is-trim* :: ('*q*, '*f*) *reg* ⇒ *bool* **where**
reg-is-trim R ≡ *ta-is-trim* (*fin R*) (*ta R*)

We obtain a trim automaton by restriction it to reachable and productive states.

abbreviation *ta-only-reach* :: ('*q*, '*f*) *ta* ⇒ ('*q*, '*f*) *ta* **where**
ta-only-reach $\mathcal{A}$ ≡ *ta-restrict* $\mathcal{A}$ (*ta-reachable* $\mathcal{A}$)

abbreviation *ta-only-prod* :: '*q fset* ⇒ ('*q*,'*f*) *ta* ⇒ ('*q*,'*f*) *ta* **where**
ta-only-prod P $\mathcal{A}$ ≡ *ta-restrict* $\mathcal{A}$ (*ta-productive P* $\mathcal{A}$)

definition *reg-reach* **where**
reg-reach R = Reg (fin R) (ta-only-reach (ta R))

definition *reg-prod* **where**
reg-prod R = Reg (fin R) (ta-only-prod (fin R) (ta R))

definition *trim-ta* :: *'q fset ⇒ ('q, 'f) ta ⇒ ('q, 'f) ta* **where**
trim-ta P 𝒜 = ta-only-prod P (ta-only-reach 𝒜)

definition *trim-reg* **where**
trim-reg R = Reg (fin R) (trim-ta (fin R) (ta R))

3.1.4 Mapping over tree automata

definition *fmap-states-ta* :: *('a ⇒ 'b) ⇒ ('a, 'f) ta ⇒ ('b, 'f) ta* **where**
fmap-states-ta f 𝒜 = TA (map-ta-rule f id |‘| rules 𝒜) (map-both f |‘| eps 𝒜)

definition *fmap-funs-ta* :: *('f ⇒ 'g) ⇒ ('a, 'f) ta ⇒ ('a, 'g) ta* **where**
fmap-funs-ta f 𝒜 = TA (map-ta-rule id f |‘| rules 𝒜) (eps 𝒜)

definition *fmap-states-reg* :: *('a ⇒ 'b) ⇒ ('a, 'f) reg ⇒ ('b, 'f) reg* **where**
fmap-states-reg f R = Reg (f |‘| fin R) (fmap-states-ta f (ta R))

definition *fmap-funs-reg* :: *('f ⇒ 'g) ⇒ ('a, 'f) reg ⇒ ('a, 'g) reg* **where**
fmap-funs-reg f R = Reg (fin R) (fmap-funs-ta f (ta R))

3.1.5 Product construction (language intersection)

definition *prod-ta-rules* :: *('q1,'f) ta ⇒ ('q2,'f) ta ⇒ ('q1 × 'q2, 'f) ta-rule fset*
where
prod-ta-rules 𝒜 ℬ = {| TA-rule f qs q | f qs q. TA-rule f (map fst qs) (fst q) |∈| rules 𝒜 ∧
TA-rule f (map snd qs) (snd q) |∈| rules ℬ|}
declare *prod-ta-rules-def* [*simp*]

definition *prod-epsLp* **where**
prod-epsLp 𝒜 ℬ = (λ (p, q). (fst p, fst q) |∈| eps 𝒜 ∧ snd p = snd q ∧ snd q |∈| 𝒬 ℬ)
definition *prod-epsRp* **where**
prod-epsRp 𝒜 ℬ = (λ (p, q). (snd p, snd q) |∈| eps ℬ ∧ fst p = fst q ∧ fst q |∈| 𝒬 𝒜)

definition *prod-ta* :: *('q1,'f) ta ⇒ ('q2,'f) ta ⇒ ('q1 × 'q2, 'f) ta* **where**
prod-ta 𝒜 ℬ = TA (prod-ta-rules 𝒜 ℬ)
(fCollect (prod-epsLp 𝒜 ℬ) |∪| fCollect (prod-epsRp 𝒜 ℬ))

definition *reg-intersect* **where**
reg-intersect R L = Reg (fin R |×| fin L) (prod-ta (ta R) (ta L))

3.1.6 Union construction (language union)

definition *ta-union* **where**
ta-union $\mathcal{A}$ $\mathcal{B}$ = *TA* (*rules* $\mathcal{A}$ |∪| *rules* $\mathcal{B}$) (*eps* $\mathcal{A}$ |∪| *eps* $\mathcal{B}$)

definition *reg-union* **where**
reg-union R L = *Reg* (*Inl* |‘| (*fin R* |∩| $\mathcal{Q}_r$ *R*) |∪| *Inr* |‘| (*fin L* |∩| $\mathcal{Q}_r$ *L*))
(*ta-union* (*fmap-states-ta Inl* (*ta R*)) (*fmap-states-ta Inr* (*ta L*)))

3.1.7 Epsilon free and tree automaton accepting empty language

definition *eps-free-rulep* **where**
eps-free-rulep $\mathcal{A}$ = (λ *r*. ∃ *f qs q q'*. *r* = *TA-rule f qs q'* ∧ *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|))

definition *eps-free* :: (*'q*, *'f*) *ta* ⇒ (*'q*, *'f*) *ta* **where**
eps-free $\mathcal{A}$ = *TA* (*fCollect* (*eps-free-rulep* $\mathcal{A}$)) {||}

definition *is-ta-eps-free* :: (*'q*, *'f*) *ta* ⇒ *bool* **where**
is-ta-eps-free $\mathcal{A}$ ⟷ *eps* $\mathcal{A}$ = {||}

definition *ta-empty* :: *'q fset* ⇒ (*'q*,*'f*) *ta* ⇒ *bool* **where**
ta-empty Q $\mathcal{A}$ ⟷ *ta-reachable* $\mathcal{A}$ |∩| *Q* |⊆| {||}

definition *eps-free-reg* **where**
eps-free-reg R = *Reg* (*fin R*) (*eps-free* (*ta R*))

definition *reg-empty* **where**
reg-empty R = *ta-empty* (*fin R*) (*ta R*)

3.1.8 Relabeling tree automaton states to natural numbers

definition *map-fset-to-nat* :: (*'a* :: *linorder*) *fset* ⇒ *'a* ⇒ *nat* **where**
map-fset-to-nat X = (λ*x*. *the* (*mem-idx x* (*sorted-list-of-fset X*)))

definition *map-fset-fset-to-nat* :: (*'a* :: *linorder*) *fset fset* ⇒ *'a fset* ⇒ *nat* **where**
map-fset-fset-to-nat X = (λ*x*. *the* (*mem-idx* (*sorted-list-of-fset x*) (*sorted-list-of-fset* (*sorted-list-of-fset* |‘| *X*))))

definition *relabel-ta* :: (*'q* :: *linorder*, *'f*) *ta* ⇒ (*nat*, *'f*) *ta* **where**
relabel-ta $\mathcal{A}$ = *fmap-states-ta* (*map-fset-to-nat* ($\mathcal{Q}$ $\mathcal{A}$)) $\mathcal{A}$

definition *relabel-*Q_f :: (*'q* :: *linorder*) *fset* ⇒ (*'q* :: *linorder*, *'f*) *ta* ⇒ *nat fset* **where**
relabel-Q_f *Q* $\mathcal{A}$ = *map-fset-to-nat* ($\mathcal{Q}$ $\mathcal{A}$) |‘| (*Q* |∩| $\mathcal{Q}$ $\mathcal{A}$)
definition *relabel-reg* :: (*'q* :: *linorder*, *'f*) *reg* ⇒ (*nat*, *'f*) *reg* **where**
relabel-reg R = *Reg* (*relabel-*Q_f (*fin R*) (*ta R*)) (*relabel-ta* (*ta R*))

— The instantiation of < and ≤ for finite sets are |⊂| and |⊆| which don't give rise to a total order and therefore it cannot be an instance of the type class linorder.

However taking the lexographic order of the sorted list of each finite set gives rise to a total order. Therefore we provide a relabeling for tree automata where the states are finite sets. This allows us to relabel the well known power set construction.

definition *relabel-fset-ta* :: (('*q* :: *linorder*) *fset*, '*f*) *ta* ⇒ (*nat*, '*f*) *ta* **where**
relabel-fset-ta $\mathcal{A}$ = *fmap-states-ta* (*map-fset-fset-to-nat* (*Q* $\mathcal{A}$)) $\mathcal{A}$

definition *relabel-fset-Q*$_f$ :: ('*q* :: *linorder*) *fset fset* ⇒ (('*q* :: *linorder*) *fset*, '*f*) *ta* ⇒ *nat fset* **where**
relabel-fset-Q$_f$ *Q* $\mathcal{A}$ = *map-fset-fset-to-nat* (*Q* $\mathcal{A}$) |`| (*Q* |∩| *Q* $\mathcal{A}$)

definition *relable-fset-reg* :: (('*q* :: *linorder*) *fset*, '*f*) *reg* ⇒ (*nat*, '*f*) *reg* **where**
relable-fset-reg R = *Reg* (*relabel-fset-Q*$_f$ (*fin R*) (*ta R*)) (*relabel-fset-ta* (*ta R*))

definition *srules* $\mathcal{A}$ = *fset* (*rules* $\mathcal{A}$)
definition *seps* $\mathcal{A}$ = *fset* (*eps* $\mathcal{A}$)

lemma *rules-transfer* [*transfer-rule*]:
rel-fun (=) (*pcr-fset* (=)) *srules rules* **unfolding** *rel-fun-def*
by (*auto simp add*: *cr-fset-def fset.pcr-cr-eq srules-def*)

lemma *eps-transfer* [*transfer-rule*]:
rel-fun (=) (*pcr-fset* (=)) *seps eps* **unfolding** *rel-fun-def*
by (*auto simp add*: *cr-fset-def fset.pcr-cr-eq seps-def*)

lemma *TA-equalityI*:
rules $\mathcal{A}$ = *rules* $\mathcal{B}$ ⟹ *eps* $\mathcal{A}$ = *eps* $\mathcal{B}$ ⟹ $\mathcal{A}$ = $\mathcal{B}$
using *ta.expand* **by** *blast*

lemma *rule-states-code* [*code*]:
rule-states Δ = |⋃| ((λ *r*. *finsert* (*r-rhs r*) (*fset-of-list* (*r-lhs-states r*))) |`| Δ)
unfolding *rule-states-def*
by *fastforce*

lemma *eps-states-code* [*code*]:
eps-states Δ$_\varepsilon$ = |⋃| ((λ (*q*,*q*'). {|*q*,*q*'|}) |`| Δ$_\varepsilon$) (**is** *?Ls* = *?Rs*)
unfolding *eps-states-def*
by (*force simp add*: *case-prod-beta*')

lemma *rule-states-empty* [*simp*]:
rule-states {||} = {||}
by (*auto simp*: *rule-states-def*)

lemma *eps-states-empty* [*simp*]:
eps-states {||} = {||}
by (*auto simp*: *eps-states-def*)

lemma *rule-states-union* [*simp*]:

rule-states (Δ |$\cup$| Γ) = *rule-states* Δ |$\cup$| *rule-states* Γ
unfolding *rule-states-def*
by *fastforce*

lemma *rule-states-mono*:
Δ |$\subseteq$| Γ $\Longrightarrow$ *rule-states* Δ |$\subseteq$| *rule-states* Γ
unfolding *rule-states-def*
by *force*

lemma *eps-states-union* [*simp*]:
eps-states (Δ |$\cup$| Γ) = *eps-states* Δ |$\cup$| *eps-states* Γ
unfolding *eps-states-def*
by *auto*

lemma *eps-states-image* [*simp*]:
eps-states (*map-both f* |`| $\Delta_\varepsilon$) = *f* |`| *eps-states* Δ_ε
unfolding *eps-states-def map-prod-def*
by (*force simp*: *fimage-iff*)

lemma *eps-states-mono*:
Δ |$\subseteq$| Γ $\Longrightarrow$ *eps-states* Δ |$\subseteq$| *eps-states* Γ
unfolding *eps-states-def*
by *transfer auto*

lemma *eps-statesI* [*intro*]:
(p, q) |$\in$| Δ $\Longrightarrow$ p |$\in$| *eps-states* Δ
(p, q) |$\in$| Δ $\Longrightarrow$ q |$\in$| *eps-states* Δ
unfolding *eps-states-def*
by (*auto simp add*: *rev-image-eqI*)

lemma *eps-statesE* [*elim*]:
assumes p |$\in$| *eps-states* Δ
obtains q **where** (p, q) |$\in$| Δ $\vee$ (q, p) |$\in$| Δ **using** *assms*
unfolding *eps-states-def*
by (*transfer*, *auto*)+

lemma *rule-statesE* [*elim*]:
assumes q |$\in$| *rule-states* Δ
obtains *f ps p* **where** *TA-rule f ps p* |$\in$| Δ q |$\in$| (*fset-of-list ps*) $\vee$ $q = p$ **using** *assms*
proof −
assume *ass*: ($\bigwedge$*f ps p*. *f ps* $\rightarrow$ *p* |$\in$| Δ $\Longrightarrow$ *q* |$\in$| *fset-of-list ps* $\vee$ *q* = *p* $\Longrightarrow$ *thesis*)
from *assms* **obtain** *r* **where** *r* |$\in$| Δ *q* |$\in$| *fset-of-list* (*r-lhs-states r*) $\vee$ *q* = *r-rhs r*
by (*auto simp*: *rule-states-def*)
then show *thesis* **using** *ass*
by (*cases r*) *auto*
qed

lemma *rule-statesI* [*intro*]:
assumes r $|\in|$ Δ q $|\in|$ *finsert* (*r-rhs r*) (*fset-of-list* (*r-lhs-states r*))
shows q $|\in|$ *rule-states* Δ **using** *assms*
by (*auto simp*: *rule-states-def*)

Destruction rule for states

lemma *rule-statesD*:
r $|\in|$ (*rules* $\mathcal{A}$) $\Longrightarrow$ *r-rhs r* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ *f qs* $\rightarrow$ q $|\in|$ (*rules* $\mathcal{A}$) $\Longrightarrow$ q $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
r $|\in|$ (*rules* $\mathcal{A}$) $\Longrightarrow$ p $|\in|$ *fset-of-list* (*r-lhs-states r*) $\Longrightarrow$ p $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
f qs $\rightarrow$ q $|\in|$ (*rules* $\mathcal{A}$) $\Longrightarrow$ p $|\in|$ *fset-of-list qs* $\Longrightarrow$ p $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
by (*force simp*: $\mathcal{Q}$*-def rule-states-def fimage-iff*)+

lemma *eps-states* [*simp*]: (*eps* $\mathcal{A}$) $|\subseteq|$ $\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathcal{A}$
unfolding $\mathcal{Q}$*-def eps-states-def rule-states-def*
by (*auto simp add*: *rev-image-eqI*)

lemma *eps-statesD*: (p, q) $|\in|$ (*eps* $\mathcal{A}$) $\Longrightarrow$ p $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ $\wedge$ q $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
using *eps-states* **by** (*auto simp add*: $\mathcal{Q}$*-def*)

lemma *eps-trancl-statesD*:
(p, q) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ $\Longrightarrow$ p $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ $\wedge$ q $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
by (*induct rule*: *ftrancl-induct*) (*auto dest*: *eps-statesD*)

lemmas *eps-dest-all* = *eps-statesD eps-trancl-statesD*

Mapping over function symbols/states

lemma *finite-Collect-ta-rule*:
finite {*TA-rule f qs q* | *f qs q*. *TA-rule f qs q* $|\in|$ *rules* $\mathcal{A}$} (**is** *finite ?S*)
proof −
have {*f qs* $\rightarrow$ *q* |*f qs q*. *f qs* $\rightarrow$ *q* $|\in|$ *rules* $\mathcal{A}$} $\subseteq$ *fset* (*rules* $\mathcal{A}$)
by *auto*
from *finite-subset*[*OF this*] **show** *?thesis* **by** *simp*
qed

lemma *map-ta-rule-finite*:
finite Δ $\Longrightarrow$ *finite* {*TA-rule* (*g h*) (*map f qs*) (*f q*) | *h qs q*. *TA-rule h qs q* $\in$ Δ}
proof (*induct rule*: *finite.induct*)
case (*insertI A a*)
have *union*: {*TA-rule* (*g h*) (*map f qs*) (*f q*) |*h qs q*. *TA-rule h qs q* $\in$ *insert a A*} =
{*TA-rule* (*g h*) (*map f qs*) (*f q*) | *h qs q*. *TA-rule h qs q* = *a*} $\cup$ {*TA-rule* (*g h*) (*map f qs*) (*f q*) |*h qs q*. *TA-rule h qs q* $\in$ *A*}
by *auto*
have *finite* {*g h map f qs* $\rightarrow$ *f q* |*h qs q*. *h qs* $\rightarrow$ *q* = *a*}
by (*cases a*) *auto*
from *finite-UnI*[*OF this insertI*(*2*)] **show** *?case* **unfolding** *union* **.**
qed *auto*

lemmas *map-ta-rule-fset-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ **for** Δ, *simplified*]

lemmas *map-ta-rule-states-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ *id* **for** Δ, *simplified*]
lemmas *map-ta-rule-funsym-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ - *id* **for** Δ, *simplified*]

lemma *map-ta-rule-comp*:
map-ta-rule f g ∘ *map-ta-rule f′ g′* = *map-ta-rule* (*f* ∘ *f′*) (*g* ∘ *g′*)
using *ta-rule.map-comp*[*of f g*]
by (*auto simp*: *comp-def*)

lemma *map-ta-rule-cases*:
map-ta-rule f g r = *TA-rule* (*g* (*r-root r*)) (*map f* (*r-lhs-states r*)) (*f* (*r-rhs r*))
by (*cases r*) *auto*

lemma *map-ta-rule-prod-swap-id* [*simp*]:
map-ta-rule prod.swap prod.swap (*map-ta-rule prod.swap prod.swap r*) = *r*
by (*auto simp*: *map-ta-rule-cases*)

lemma *rule-states-image* [*simp*]:
rule-states (*map-ta-rule f g* |‘| Δ) = *f* |‘| *rule-states* Δ (**is** *?Ls* = *?Rs*)
proof −
{fix *q* **assume** *q* |∈| *?Ls*
then obtain *r* **where** *r* |∈| Δ
q |∈| *finsert* (*r-rhs* (*map-ta-rule f g r*)) (*fset-of-list* (*r-lhs-states* (*map-ta-rule f g r*)))
by (*auto simp*: *rule-states-def*)
then have *q* |∈| *?Rs* **by** (*cases r*) (*force simp*: *fimage-iff*)**}**
moreover
{fix *q* **assume** *q* |∈| *?Rs*
then obtain *r p* **where** *r* |∈| Δ *f p* = *q*
p |∈| *finsert* (*r-rhs r*) (*fset-of-list* (*r-lhs-states r*))
by (*auto simp*: *rule-states-def*)
then have *q* |∈| *?Ls* **by** (*cases r*) (*force simp*: *fimage-iff*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Q-mono*:
(*rules* $\mathcal{A}$) |⊆| (*rules* $\mathcal{B}$) ⟹ (*eps* $\mathcal{A}$) |⊆| (*eps* $\mathcal{B}$) ⟹ $\mathcal{Q}$ $\mathcal{A}$ |⊆| $\mathcal{Q}$ $\mathcal{B}$
using *rule-states-mono eps-states-mono* **unfolding** *Q-def*
by *blast*

lemma *Q-subseteq-I*:
assumes ⋀ *r*. *r* |∈| *rules* $\mathcal{A}$ ⟹ *r-rhs r* |∈| *S*
and ⋀ *r*. *r* |∈| *rules* $\mathcal{A}$ ⟹ *fset-of-list* (*r-lhs-states r*) |⊆| *S*
and ⋀ *e*. *e* |∈| *eps* $\mathcal{A}$ ⟹ *fst e* |∈| *S* ∧ *snd e* |∈| *S*
shows $\mathcal{Q}$ $\mathcal{A}$ |⊆| *S* **using** *assms* **unfolding** *Q-def*
by (*auto simp*: *rule-states-def*) *blast*

lemma *finite-states*:
finite {*q*. ∃ *f p ps*. *f ps* → *p* |∈| *rules* $\mathcal{A}$ ∧ (*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|)} (**is** *finite ?set*)
proof −
have *?set* ⊆ *fset* ($\mathcal{Q}$ $\mathcal{A}$)
by (*intro subsetI*, *drule CollectD*)
(*metis eps-trancl-statesD rule-statesD*(*2*))
from *finite-subset*[*OF this*] **show** *?thesis* **by** *auto*
qed

Collecting all states reachable from target of rules

lemma *finite-ta-rhs-states* [*simp*]:
finite {*q*. ∃*p*. *p* |∈| *rule-target-states* (*rules* $\mathcal{A}$) ∧ (*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|)}
(**is** *finite ?Set*)
proof −
have *?Set* ⊆ *fset* ($\mathcal{Q}$ $\mathcal{A}$)
by (*auto dest*: *rule-statesD*)
(*metis eps-trancl-statesD rule-statesD*(*1*))+
from *finite-subset*[*OF this*] **show** *?thesis*
by *auto*
qed

Computing the signature induced by the rule set of given tree automaton

lemma *ta-sigI* [*intro*]:
TA-rule f qs q |∈| (*rules* $\mathcal{A}$) ⟹ *length qs* = *n* ⟹ (*f*, *n*) |∈| *ta-sig* $\mathcal{A}$ **unfolding** *ta-sig-def*
using *mk-disjoint-finsert* **by** *fastforce*

lemma *ta-sig-mono*:
(*rules* $\mathcal{A}$) |⊆| (*rules* $\mathcal{B}$) ⟹ *ta-sig* $\mathcal{A}$ |⊆| *ta-sig* $\mathcal{B}$
by (*auto simp*: *ta-sig-def*)

lemma *finite-eps*:
finite {*q*. ∃ *f ps p*. *f ps* → *p* |∈| *rules* $\mathcal{A}$ ∧ (*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|)} (**is** *finite ?S*)
by (*intro finite-subset*[*OF - finite-ta-rhs-states*[*of* $\mathcal{A}$]]) (*auto intro*!: *bexI*)

lemma *collect-snd-trancl-fset*:
{*p*. (*q*, *p*) |∈| (*eps* $\mathcal{A}$)|⁺|} = *fset* (*snd* |‘| (*ffilter* (λ *x*. *fst x* = *q*) ((*eps* $\mathcal{A}$)|⁺|)))
by (*auto simp*: *image-iff*) *force*

lemma *ta-der-Var*:
q |∈| *ta-der* $\mathcal{A}$ (*Var x*) ⟷ *x* = *q* ∨ (*x*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|
by (*auto simp*: *collect-snd-trancl-fset*)

lemma *ta-der-Fun*:
q |∈| *ta-der* $\mathcal{A}$ (*Fun f ts*) ⟷ (∃ *ps p*. *TA-rule f ps p* |∈| (*rules* $\mathcal{A}$) ∧
(*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|) ∧ *length ps* = *length ts* ∧
(∀ *i* < *length ts*. *ps* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*))) (**is** *?Ls* ⟷ *?Rs*)

unfolding *ta-der.simps*
by (*intro iffI fCollect-memberI finite-Collect-less-eq*[*OF - finite-eps*[*of* $\mathcal{A}$]]) *auto*

declare *ta-der.simps*[*simp del*]
declare *ta-der.simps*[*code del*]
lemmas *ta-der-simps* [*simp*] = *ta-der-Var ta-der-Fun*

lemma *ta-der′-Var*:
Var q |∈| *ta-der′* $\mathcal{A}$ (*Var x*) ⟷ $x = q \vee (x, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$|
by (*auto simp*: *collect-snd-trancl-fset*)

lemma *ta-der′-Fun*:
Var q |∈| *ta-der′* $\mathcal{A}$ (*Fun f ts*) ⟷ *q* |∈| *ta-der* $\mathcal{A}$ (*Fun f ts*)
unfolding *ta-der′.simps*
by (*intro iffI funionI1 fCollect-memberI*)
(*auto simp del*: *ta-der-Fun ta-der-Var simp*: *fset-image-conv*)

lemma *ta-der′-Fun2*:
Fun f ps |∈| *ta-der′* $\mathcal{A}$ (*Fun g ts*) ⟷ $f = g \wedge$ *length ps* = *length ts* $\wedge$ ($\forall$ *i*<*length ts*. *ps* ! *i* |∈| *ta-der′* $\mathcal{A}$ (*ts* ! *i*))
proof −
have *f*: *finite* {*ss*. *set ss* ⊆ *fset* (|⋃| (*fset-of-list* (*map* (*ta-der′* $\mathcal{A}$) *ts*))) ∧ *length ss* = *length ts*}
by (*intro finite-lists-length-eq*) *auto*
have *finite* {*ss*. *length ss* = *length ts* ∧ ($\forall$ *i*<*length ts*. *ss* ! *i* |∈| *ta-der′* $\mathcal{A}$ (*ts* ! *i*))}
by (*intro finite-subset*[*OF - f*])
(*force simp*: *in-fset-conv-nth simp flip*: *fset-of-list-elem*)
then show *?thesis* **unfolding** *ta-der′.simps*
by (*intro iffI funionI2 fCollect-memberI*)
(*auto simp del*: *ta-der-Fun ta-der-Var*)
qed

declare *ta-der′.simps*[*simp del*]
declare *ta-der′.simps*[*code del*]
lemmas *ta-der′-simps* [*simp*] = *ta-der′-Var ta-der′-Fun ta-der′-Fun2*

Induction schemes for the most used cases

lemma *ta-der-induct*[*consumes 1*, *case-names Var Fun*]:
assumes *reach*: *q* |∈| *ta-der* $\mathcal{A}$ *t*
and *VarI*: $\bigwedge$ *q v*. $v = q \vee (v, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *P* (*Var v*) *q*
and *FunI*: $\bigwedge$*f ts ps p q*. *f ps* → *p* |∈| *rules* $\mathcal{A}$ ⟹ *length ts* = *length ps* ⟹ $p = q \vee (p, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹
($\bigwedge$*i*. *i* < *length ts* ⟹ *ps* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*)) ⟹
($\bigwedge$*i*. *i* < *length ts* ⟹ *P* (*ts* ! *i*) (*ps* ! *i*)) ⟹ *P* (*Fun f ts*) *q*
shows *P t q* **using** *assms*(*1*)
by (*induct t arbitrary*: *q*) (*auto simp*: *VarI FunI*)

lemma *ta-der-gterm-induct*[*consumes 1*, *case-names GFun*]:

assumes *reach*: $q \mid\in\mid$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
and *Fun*: $\bigwedge f\ ts\ ps\ p\ q.$ *TA-rule f ps p* $\mid\in\mid$ *rules* $\mathcal{A} \Longrightarrow$ *length ts* = *length ps* $\Longrightarrow$ $p = q \vee (p,\ q) \mid\in\mid (eps\ \mathcal{A})^{+}\mid \Longrightarrow$
$(\bigwedge i.\ i < length\ ts \Longrightarrow ps\ !\ i \mid\in\mid ta\text{-}der\ \mathcal{A}\ (term\text{-}of\text{-}gterm\ (ts\ !\ i))) \Longrightarrow$
$(\bigwedge i.\ i < length\ ts \Longrightarrow P\ (ts\ !\ i)\ (ps\ !\ i)) \Longrightarrow P\ (GFun\ f\ ts)\ q$
shows *P t q* **using** *assms*(*1*)
by (*induct t arbitrary*: *q*) (*auto simp*: *Fun*)

lemma *ta-der-rule-empty*:
assumes $q \mid\in\mid$ *ta-der* (*TA* $\{\mid\mid\}$ Δ_ε) *t*
obtains *p* **where** *t* = *Var p p* = *q* $\vee$ $(p,\ q) \mid\in\mid \Delta_\varepsilon\mid^{+}\mid$
using *assms* **by** (*cases t*) *auto*

lemma *ta-der-eps*:
assumes $(p,\ q) \mid\in\mid$ (*eps* $\mathcal{A}$) **and** *p* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t*
shows *q* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t* **using** *assms*
by (*cases t*) (*auto intro*: *ftrancl-into-trancl*)

lemma *ta-der-trancl-eps*:
assumes $(p,\ q) \mid\in\mid (eps\ \mathcal{A})\mid^{+}\mid$ **and** *p* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t*
shows *q* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t* **using** *assms*
by (*induct rule*: *ftrancl-induct*) (*auto intro*: *ftrancl-into-trancl ta-der-eps*)

lemma *ta-der-mono*:
(*rules* $\mathcal{A}$) $\mid\subseteq\mid$ (*rules* $\mathcal{B}$) $\Longrightarrow$ (*eps* $\mathcal{A}$) $\mid\subseteq\mid$ (*eps* $\mathcal{B}$) $\Longrightarrow$ *ta-der* $\mathcal{A}$ *t* $\mid\subseteq\mid$ *ta-der* $\mathcal{B}$ *t*
proof (*induct t*)
case (*Var x*) **then show** *?case*
by (*auto dest*: *ftrancl-mono*[*of* - *eps* $\mathcal{A}$ *eps* $\mathcal{B}$])
next
case (*Fun f ts*)
show *?case* **using** *Fun*(*1*)[*OF nth-mem Fun*(*2*, *3*)]
by (*auto dest*!: *fsubsetD*[*OF Fun*(*2*)] *ftrancl-mono*[*OF* - *Fun*(*3*)]) *blast*+
qed

lemma *ta-der-el-mono*:
(*rules* $\mathcal{A}$) $\mid\subseteq\mid$ (*rules* $\mathcal{B}$) $\Longrightarrow$ (*eps* $\mathcal{A}$) $\mid\subseteq\mid$ (*eps* $\mathcal{B}$) $\Longrightarrow$ *q* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t* $\Longrightarrow$ *q* $\mid\in\mid$ *ta-der* $\mathcal{B}$ *t*
using *ta-der-mono* **by** *blast*

lemma *ta-der′-ta-der*:
assumes *t* $\mid\in\mid$ *ta-der′* $\mathcal{A}$ *s p* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *t*
shows *p* $\mid\in\mid$ *ta-der* $\mathcal{A}$ *s* **using** *assms*
proof (*induction arbitrary*: *p t rule*: *ta-der′.induct*)
case (*2* $\mathcal{A}$ *f ts*) **show** *?case* **using** *2*(*2*−)
proof (*induction t*)
case (*Var x*) **then show** *?case*
by *auto* (*meson ftrancl-trans*)
next
case (*Fun g ss*)

```
    have ss-props: g = f length ss = length ts ∀ i < length ts. ss ! i |∈| ta-der' 𝒜
(ts ! i)
      using Fun(2) by auto
    then show ?thesis using Fun(1)[OF nth-mem] Fun(2−)
      by (auto simp: ss-props)
        (metis (no-types, lifting) 2.IH ss-props(3))+
  qed
qed (auto dest: ftrancl-trans simp: ta-der'.simps)

lemma ta-der'-empty:
  assumes t |∈| ta-der' (TA {||} {||}) s
  shows t = s using assms
  by (induct s arbitrary: t) (auto simp add: ta-der'.simps nth-equalityI)

lemma ta-der'-to-ta-der:
  Var q |∈| ta-der' 𝒜 s ⟹ q |∈| ta-der 𝒜 s
  using ta-der'-ta-der by fastforce

lemma ta-der-to-ta-der':
  q |∈| ta-der 𝒜 s ⟷ Var q |∈| ta-der' 𝒜 s
  by (induct s arbitrary: q) auto

lemma ta-der'-poss:
  assumes t |∈| ta-der' 𝒜 s
  shows poss t ⊆ poss s using assms
proof (induct s arbitrary: t)
  case (Fun f ts)
  show ?case using Fun(2) Fun(1)[OF nth-mem, of i args t ! i for i]
    by (cases t) auto
qed (auto simp: ta-der'.simps)

lemma ta-der'-refl[simp]: t |∈| ta-der' 𝒜 t
  by (induction t) fastforce+

lemma ta-der'-eps:
  assumes Var p |∈| ta-der' 𝒜 s and (p, q) |∈| (eps 𝒜)|^+|
  shows Var q |∈| ta-der' 𝒜 s using assms
  by (cases s, auto dest: ftrancl-trans) (meson ftrancl-trans)

lemma ta-der'-trans:
  assumes t |∈| ta-der' 𝒜 s and u |∈| ta-der' 𝒜 t
  shows u |∈| ta-der' 𝒜 s using assms
proof (induct t arbitrary: u s)
  case (Fun f ts) note IS = Fun(2−) note IH = Fun(1)[OF nth-mem, of i args
s ! i for i]
  show ?case
  proof (cases s)
    case (Var x1)
    then show ?thesis using IS by (auto simp: ta-der'.simps)
```

next
case [*simp*]: (*Fun g ss*)
show *?thesis* **using** *IS IH*
by (*cases u*, *auto*) (*metis ta-der-to-ta-der'*)+
qed
qed (*auto simp*: *ta-der'.simps ta-der'-eps*)

Connecting contexts to derivation definition

lemma *ta-der-ctxt*:
assumes *p*: *p* |∈| *ta-der* $\mathcal{A}$ *t q* |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ p\rangle$
shows *q* |∈| *ta-der* $\mathcal{A}$ $C\langle t\rangle$ **using** *assms*(*2*)
proof (*induct C arbitrary*: *q*)
case *Hole* **then show** *?case* **using** *assms*
by (*auto simp*: *ta-der-trancl-eps*)
next
case (*More f ss C ts*)
from *More*(*2*) **obtain** *qs r* **where**
rule: *f qs* → *r* |∈| *rules* $\mathcal{A}$ *length qs* = *Suc* (*length ss* + *length ts*) **and**
reach: ∀ *i* < *Suc* (*length ss* + *length ts*). *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ ((*ss* @ $C\langle Var\ p\rangle$ # *ts*) ! *i*) *r* = *q* ∨ (*r*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|
by *auto*
have *i* < *Suc* (*length ss* + *length ts*) ⟹ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ ((*ss* @ $C\langle t\rangle$ # *ts*) ! *i*) **for** *i*
using *More*(*1*)[*of qs* ! *length ss*] *assms rule*(*2*) *reach*(*1*)
unfolding *nth-append-Cons* **by** *presburger*
then show *?case* **using** *rule reach*(*2*) **by** *auto*
qed

lemma *ta-der-eps-ctxt*:
assumes *p* |∈| *ta-der A* $C\langle Var\ q'\rangle$ **and** (*q*, *q'*) |∈| (*eps A*)|$^+$|
shows *p* |∈| *ta-der A* $C\langle Var\ q\rangle$
using *assms* **by** (*meson ta-der-Var ta-der-ctxt*)

lemma *rule-reachable-ctxt-exist*:
assumes *rule*: *f qs* → *q* |∈| *rules* $\mathcal{A}$ **and** *i* < *length qs*
shows ∃ *C*. *q* |∈| *ta-der* $\mathcal{A}$ (*C* ⟨*Var* (*qs* ! *i*)⟩) **using** *assms*
by (*intro exI*[*of* - *More f* (*map Var* (*take i qs*)) □ (*map Var* (*drop* (*Suc i*) *qs*))])
(*auto simp*: *min-def nth-append-Cons intro*!: *exI*[*of* - *q*] *exI*[*of* - *qs*])

lemma *ta-der-ctxt-decompose*:
assumes *q* |∈| *ta-der* $\mathcal{A}$ $C\langle t\rangle$
shows ∃ *p* . *p* |∈| *ta-der* $\mathcal{A}$ *t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ p\rangle$ **using** *assms*
proof (*induct C arbitrary*: *q*)
case (*More f ss C ts*)
from *More*(*2*) **obtain** *qs r* **where**
rule: *f qs* → *r* |∈| *rules* $\mathcal{A}$ *length qs* = *Suc* (*length ss* + *length ts*) **and**
reach: ∀ *i* < *Suc* (*length ss* + *length ts*). *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ ((*ss* @ $C\langle t\rangle$ # *ts*) ! *i*)
r = *q* ∨ (*r*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|

by *auto*
obtain *p* **where** *p*: *p* |∈| *ta-der* $\mathcal{A}$ *t qs* ! *length ss* |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ p\rangle$
using *More*(*1*)[*of qs* ! *length ss*] *reach*(*1*) *rule*(*2*)
by (*metis less-add-Suc1 nth-append-length*)
have *i* < *Suc* (*length ss* + *length ts*) ⟹ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ ((*ss* @ $C\langle Var\ p\rangle$ # *ts*) ! *i*) **for** *i*
using *reach rule*(*2*) *p* **by** (*auto simp*: *p*(*2*) *nth-append-Cons*)
then have *q* |∈| *ta-der* $\mathcal{A}$ (*More f ss C ts*)$\langle Var\ p\rangle$ **using** *rule reach*
by *auto*
then show *?case* **using** *p*(*1*) **by** (*intro exI*[*of* - *p*]) *blast*
qed *auto*

— Relation between reachable states and states of a tree automaton

lemma *ta-der-states*:
ta-der $\mathcal{A}$ *t* |⊆| $\mathcal{Q}$ $\mathcal{A}$ |∪| *fvars-term t*
proof (*induct t*)
case (*Var x*) **then show** *?case*
by (*auto simp*: *eq-onp-same-args*)
(*metis eps-trancl-statesD*)
case (*Fun f ts*) **then show** *?case*
by (*auto simp*: *rule-statesD*(*2*) *eps-trancl-statesD*)
qed

lemma *ground-ta-der-states*:
ground t ⟹ *ta-der* $\mathcal{A}$ *t* |⊆| $\mathcal{Q}$ $\mathcal{A}$
using *ta-der-states*[*of* $\mathcal{A}$ *t*] **by** *auto*

lemmas *ground-ta-der-statesD* = *fsubsetD*[*OF ground-ta-der-states*]

lemma *gterm-ta-der-states* [*simp*]:
q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟹ *q* |∈| $\mathcal{Q}$ $\mathcal{A}$
by (*intro ground-ta-der-states*[*THEN fsubsetD*, *of term-of-gterm t*]) *simp*

lemma *ta-der-states′*:
q |∈| *ta-der* $\mathcal{A}$ *t* ⟹ *q* |∈| $\mathcal{Q}$ $\mathcal{A}$ ⟹ *fvars-term t* |⊆| $\mathcal{Q}$ $\mathcal{A}$
proof (*induct rule*: *ta-der-induct*)
case (*Fun f ts ps p r*)
then have *i* < *length ts* ⟹ *fvars-term* (*ts* ! *i*) |⊆| $\mathcal{Q}$ $\mathcal{A}$ **for** *i*
by (*auto simp*: *in-fset-conv-nth dest*!: *rule-statesD*(*3*))
then show *?case* **by** (*force simp*: *in-fset-conv-nth*)
qed (*auto simp*: *eps-trancl-statesD*)

lemma *ta-der-not-stateD*:
q |∈| *ta-der* $\mathcal{A}$ *t* ⟹ *q* |∉| $\mathcal{Q}$ $\mathcal{A}$ ⟹ *t* = *Var q*
using *fsubsetD*[*OF ta-der-states*, *of q* $\mathcal{A}$ *t*]
by (*cases t*) (*auto dest*: *rule-statesD eps-trancl-statesD*)

lemma *ta-der-is-fun-stateD*:

is-Fun t $\Longrightarrow$ *q* $|\in|$ *ta-der* $\mathcal{A}$ *t* $\Longrightarrow$ *q* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
using *ta-der-not-stateD*[*of q* $\mathcal{A}$ *t*]
by (*cases t*) *auto*

lemma *ta-der-is-fun-fvars-stateD*:
is-Fun t $\Longrightarrow$ *q* $|\in|$ *ta-der* $\mathcal{A}$ *t* $\Longrightarrow$ *fvars-term t* $|\subseteq|$ $\mathcal{Q}$ $\mathcal{A}$
using *ta-der-is-fun-stateD*[*of t q* $\mathcal{A}$]
using *ta-der-states'*[*of q* $\mathcal{A}$ *t*]
by (*cases t*) *auto*

lemma *ta-der-not-reach*:
assumes $\bigwedge$ *r*. *r* $|\in|$ *rules* $\mathcal{A}$ $\Longrightarrow$ *r-rhs r* $\neq$ *q*
and $\bigwedge$ *e*. *e* $|\in|$ *eps* $\mathcal{A}$ $\Longrightarrow$ *snd e* $\neq$ *q*
shows *q* $|\notin|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **using** *assms*
by (*cases t*) (*fastforce dest*!: *assms*(*1*) *ftranclD2*[*of* - *q*])

lemma *ta-rhs-states-subset-states*: *ta-rhs-states* $\mathcal{A}$ $|\subseteq|$ $\mathcal{Q}$ $\mathcal{A}$
by (*auto simp*: *ta-rhs-states-def dest*: *rtranclD rule-statesD eps-trancl-statesD*)

lemma *ta-rhs-states-res*: **assumes** *is-Fun t*
shows *ta-der* $\mathcal{A}$ *t* $|\subseteq|$ *ta-rhs-states* $\mathcal{A}$
proof
fix *q* **assume** *q*: *q* $|\in|$ *ta-der* $\mathcal{A}$ *t*
from ‹*is-Fun t*› **obtain** *f ts* **where** *t*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
from *q*[*unfolded t*] **obtain** *q′ qs* **where** *TA-rule f qs q′* $|\in|$ *rules* $\mathcal{A}$
and *q*: *q′* = *q* $\vee$ (*q′*, *q*) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ **by** *auto*
then show *q* $|\in|$ *ta-rhs-states* $\mathcal{A}$ **unfolding** *ta-rhs-states-def*
by (*auto intro*!: *bexI*)
qed

Reachable states of ground terms are preserved over the *adapt-vars* function

lemma *ta-der-adapt-vars-ground* [*simp*]:
ground t $\Longrightarrow$ *ta-der A* (*adapt-vars t*) = *ta-der A t*
by (*induct t*) *auto*

lemma *gterm-of-term-inv′*:
ground t $\Longrightarrow$ *term-of-gterm* (*gterm-of-term t*) = *adapt-vars t*
by (*induct t*) (*auto 0 0 intro*!: *nth-equalityI*)

lemma *map-vars-term-term-of-gterm*:
map-vars-term f (*term-of-gterm t*) = *term-of-gterm t*
by (*induct t*) *auto*

lemma *adapt-vars-term-of-gterm*:
adapt-vars (*term-of-gterm t*) = *term-of-gterm t*
by (*induct t*) *auto*

lemma *ta-der-term-sig*:
q |∈| *ta-der* $\mathcal{A}$ *t* ⟹ *ffunas-term t* |⊆| *ta-sig* $\mathcal{A}$
proof (*induct rule*: *ta-der-induct*)
case (*Fun f ts ps p q*)
show *?case* **using** *Fun*(*1* − *4*) *Fun*(*5*)[*THEN fsubsetD*]
by (*auto simp*: *in-fset-conv-nth*)
qed *auto*

lemma *ta-der-gterm-sig*:
q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟹ *ffunas-gterm t* |⊆| *ta-sig* $\mathcal{A}$
using *ta-der-term-sig ffunas-term-of-gterm-conv*
by *fastforce*

ta-lang for terms with arbitrary variable type

lemma *ta-langE*: **assumes** *t* ∈ *ta-lang Q* $\mathcal{A}$
obtains *t′ q* **where** *ground t′ q* |∈| *Q q* |∈| *ta-der* $\mathcal{A}$ *t′ t* = *adapt-vars t′*
using *assms* **unfolding** *ta-lang-def* **by** *blast*

lemma *ta-langI*: **assumes** *ground t′ q* |∈| *Q q* |∈| *ta-der* $\mathcal{A}$ *t′ t* = *adapt-vars t′*
shows *t* ∈ *ta-lang Q* $\mathcal{A}$
using *assms* **unfolding** *ta-lang-def* **by** *blast*

lemma *ta-lang-def2*: (*ta-lang Q* ($\mathcal{A}$ :: ('*q*,'*f*)*ta*) :: ('*f*,'*v*)*terms*) = {*t*. *ground t* ∧
Q |∩| *ta-der* $\mathcal{A}$ (*adapt-vars t*) ≠ {||}}
by (*auto elim*!: *ta-langE*) (*metis adapt-vars-adapt-vars ground-adapt-vars ta-langI*)

ta-lang for *gterms*

lemma *ta-lang-to-gta-lang* [*simp*]:
ta-lang Q $\mathcal{A}$ = *term-of-gterm* ‘ *gta-lang Q* $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
proof −
{fix *t* **assume** *t* ∈ *?Ls*
from *ta-langE*[*OF this*] **obtain** *q t′* **where** *ground t′ q* |∈| *Q q* |∈| *ta-der* $\mathcal{A}$ *t′*
t = *adapt-vars t′*
by *blast*
then have *t* ∈ *?Rs* **unfolding** *gta-lang-def gta-der-def*
by (*auto simp*: *image-iff gterm-of-term-inv′ intro*!: *exI*[*of - gterm-of-term t′*])**}**
moreover
{fix *t* **assume** *t* ∈ *?Rs* **then have** *t* ∈ *?Ls*
using *ta-langI*[*OF ground-term-of-gterm - - gterm-of-term-inv′*[*OF ground-term-of-gterm*]]
by (*force simp*: *gta-lang-def gta-der-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *term-of-gterm-in-ta-lang-conv*:
term-of-gterm t ∈ *ta-lang Q* $\mathcal{A}$ ⟷ *t* ∈ *gta-lang Q* $\mathcal{A}$
by (*metis* (*mono-tags*, *lifting*) *image-iff ta-lang-to-gta-lang term-of-gterm-inv*)

lemma *gta-lang-def-sym*:
gterm-of-term ‘ ta-lang Q A = gta-lang Q A

unfolding *gta-lang-def image-def*
by (*intro Collect-cong*) (*simp add*: *gta-lang-def*)

lemma *gta-langI* [*intro*]:
assumes *q* |∈| *Q* **and** *q* |∈| *ta-der A* (*term-of-gterm t*)
shows *t* ∈ *gta-lang Q A* **using** *assms*
by (*metis adapt-vars-term-of-gterm ground-term-of-gterm ta-langI term-of-gterm-in-ta-lang-conv*)

lemma *gta-langE* [*elim*]:
assumes *t* ∈ *gta-lang Q A*
obtains *q* **where** *q* |∈| *Q* **and** *q* |∈| *ta-der A* (*term-of-gterm t*) **using** *assms*
by (*metis adapt-vars-adapt-vars adapt-vars-term-of-gterm ta-langE term-of-gterm-in-ta-lang-conv*)

lemma *gta-lang-mono*:
assumes ⋀ *t*. *ta-der A t* |⊆| *ta-der B t* **and** Q_A |⊆| Q_B
shows *gta-lang* Q_A *A* ⊆ *gta-lang* Q_B *B*
using *assms* **by** (*auto elim*!: *gta-langE intro*!: *gta-langI*)

lemma *gta-lang-term-of-gterm* [*simp*]:
term-of-gterm t ∈ *term-of-gterm ‘ gta-lang Q A* ⟷ *t* ∈ *gta-lang Q A*
by (*auto elim*!: *gta-langE intro*!: *gta-langI*) (*metis term-of-gterm-inv*)

lemma *gta-lang-subset-rules-funas*:
gta-lang Q A ⊆ $\mathcal{T}_G$ (*fset* (*ta-sig A*))
using *ta-der-gterm-sig*[*THEN fsubsetD*]
by (*force simp*: $\mathcal{T}_G$*-equivalent-def ffunas-gterm.rep-eq*)

lemma *reg-funas*:
$\mathcal{L}$ *A* ⊆ $\mathcal{T}_G$ (*fset* (*ta-sig* (*ta A*))) **using** *gta-lang-subset-rules-funas*
by (*auto simp*: $\mathcal{L}$*-def*)

lemma *ta-syms-lang*: *t* ∈ *ta-lang Q A* ⟹ *ffunas-term t* |⊆| *ta-sig A*
using *gta-lang-subset-rules-funas ffunas-gterm-gterm-of-term ta-der-gterm-sig ta-lang-def2*
by *fastforce*

lemma *gta-lang-Rest-states-conv*:
gta-lang Q A = *gta-lang* (*Q* |∩| *Q A*) *A*
by (*auto elim*!: *gta-langE*)

lemma *reg-Rest-fin-states* [*simp*]:
$\mathcal{L}$ (*reg-Restr-*Q_f *A*) = $\mathcal{L}$ *A*
using *gta-lang-Rest-states-conv*
by (*auto simp*: $\mathcal{L}$*-def reg-Restr-*Q_f*-def*)

Deterministic tree automatons

definition *ta-det* :: $('q,'f)$ *ta* $\Rightarrow$ *bool* **where**
ta-det $\mathcal{A} \longleftrightarrow$ *eps* $\mathcal{A} = \{||\} \wedge$
$(\forall$ *f qs q q'*. *TA-rule f qs q* $|\in|$ *rules* $\mathcal{A} \longrightarrow$ *TA-rule f qs q'* $|\in|$ *rules* $\mathcal{A} \longrightarrow q = q')$

definition *ta-subset* $\mathcal{A}$ $\mathcal{B} \longleftrightarrow$ *rules* $\mathcal{A}$ $|\subseteq|$ *rules* $\mathcal{B} \wedge$ *eps* $\mathcal{A}$ $|\subseteq|$ *eps* $\mathcal{B}$

lemma *ta-detE*[*elim*, *consumes 1*]: **assumes** *det*: *ta-det* $\mathcal{A}$
shows q $|\in|$ *ta-der* $\mathcal{A}$ $t \Longrightarrow q'$ $|\in|$ *ta-der* $\mathcal{A}$ $t \Longrightarrow q = q'$ **using** *assms*
by (*induct t arbitrary*: q q') (*auto simp*: *ta-det-def*, *metis nth-equalityI nth-mem*)

lemma *ta-subset-states*: *ta-subset* $\mathcal{A}$ $\mathcal{B} \Longrightarrow \mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ $\mathcal{Q}$ $\mathcal{B}$
using $\mathcal{Q}$*-mono* **by** (*auto simp*: *ta-subset-def*)

lemma *ta-subset-refl*[*simp*]: *ta-subset* $\mathcal{A}$ $\mathcal{A}$
unfolding *ta-subset-def* **by** *auto*

lemma *ta-subset-trans*: *ta-subset* $\mathcal{A}$ $\mathcal{B} \Longrightarrow$ *ta-subset* $\mathcal{B}$ $\mathfrak{C} \Longrightarrow$ *ta-subset* $\mathcal{A}$ $\mathfrak{C}$
unfolding *ta-subset-def* **by** *auto*

lemma *ta-subset-det*: *ta-subset* $\mathcal{A}$ $\mathcal{B} \Longrightarrow$ *ta-det* $\mathcal{B} \Longrightarrow$ *ta-det* $\mathcal{A}$
unfolding *ta-det-def ta-subset-def* **by** *blast*

lemma *ta-der-mono'*: *ta-subset* $\mathcal{A}$ $\mathcal{B} \Longrightarrow$ *ta-der* $\mathcal{A}$ t $|\subseteq|$ *ta-der* $\mathcal{B}$ t
using *ta-der-mono* **unfolding** *ta-subset-def* **by** *auto*

lemma *ta-lang-mono'*: *ta-subset* $\mathcal{A}$ $\mathcal{B} \Longrightarrow$ $Q_{\mathcal{A}}$ $|\subseteq|$ $Q_{\mathcal{B}} \Longrightarrow$ *ta-lang* $Q_{\mathcal{A}}$ $\mathcal{A} \subseteq$ *ta-lang* $Q_{\mathcal{B}}$ $\mathcal{B}$
using *gta-lang-mono*[*of* $\mathcal{A}$ $\mathcal{B}$] *ta-der-mono'*[*of* $\mathcal{A}$ $\mathcal{B}$]
by *auto blast*

lemma *ta-restrict-subset*: *ta-subset* (*ta-restrict* $\mathcal{A}$ Q) $\mathcal{A}$
unfolding *ta-subset-def ta-restrict-def*
by *auto*

lemma *ta-restrict-states-Q*: $\mathcal{Q}$ (*ta-restrict* $\mathcal{A}$ Q) $|\subseteq|$ Q
by (*auto simp*: $\mathcal{Q}$*-def ta-restrict-def rule-states-def eps-states-def dest*!: *fsubsetD*)

lemma *ta-restrict-states*: $\mathcal{Q}$ (*ta-restrict* $\mathcal{A}$ Q) $|\subseteq|$ $\mathcal{Q}$ $\mathcal{A}$
using *ta-subset-states*[*OF ta-restrict-subset*] **by** *fastforce*

lemma *ta-restrict-states-eq-imp-eq* [*simp*]:
assumes *eq*: $\mathcal{Q}$ (*ta-restrict* $\mathcal{A}$ Q) = $\mathcal{Q}$ $\mathcal{A}$
shows *ta-restrict* $\mathcal{A}$ $Q = \mathcal{A}$ **using** *assms*
apply (*auto simp*: *ta-restrict-def*
intro!: *ta.expand finite-subset*[*OF - finite-Collect-ta-rule*, *of - A*])

apply (*metis* (*no-types*, *lifting*) *eq fsubsetD fsubsetI rule-statesD*(*1*) *rule-statesD*(*4*) *ta-restrict-states-Q ta-rule.collapse*)
apply (*metis eps-statesD eq fin-mono ta-restrict-states-Q*)
by (*metis eps-statesD eq fsubsetD ta-restrict-states-Q*)

lemma *ta-der-ta-derict-states*:
fvars-term t |⊆| *Q* ⟹ *q* |∈| *ta-der* (*ta-restrict* $\mathcal{A}$ *Q*) *t* ⟹ *q* |∈| *Q*
by (*induct t arbitrary*: *q*) (*auto simp*: *ta-restrict-def elim*: *ftranclE*)

lemma *ta-derict-ruleI* [*intro*]:
TA-rule f qs q |∈| *rules* $\mathcal{A}$ ⟹ *fset-of-list qs* |⊆| *Q* ⟹ *q* |∈| *Q* ⟹ *TA-rule f qs q* |∈| *rules* (*ta-restrict* $\mathcal{A}$ *Q*)
by (*auto simp*: *ta-restrict-def intro*!: *ta.expand finite-subset*[*OF - finite-Collect-ta-rule*, *of - A*])

Reachable and productive states: There always is a trim automaton

lemma *finite-ta-reachable* [*simp*]:
finite {*q*. ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t*}
proof −
have {*q*. ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t*} ⊆ *fset* ($\mathcal{Q}$ $\mathcal{A}$)
using *ground-ta-der-states*[*of - A*]
by *auto*
from *finite-subset*[*OF this*] **show** *?thesis* **by** *auto*
qed

lemma *ta-reachable-states*:
ta-reachable $\mathcal{A}$ |⊆| $\mathcal{Q}$ $\mathcal{A}$
unfolding *ta-reachable-def* **using** *ground-ta-der-states*
by *force*

lemma *ta-reachableE*:
assumes *q* |∈| *ta-reachable* $\mathcal{A}$
obtains *t* **where** *ground t q* |∈| *ta-der* $\mathcal{A}$ *t*
using *assms*[*unfolded ta-reachable-def*] **by** *auto*

lemma *ta-reachable-gtermE* [*elim*]:
assumes *q* |∈| *ta-reachable* $\mathcal{A}$
obtains *t* **where** *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
using *ta-reachableE*[*OF assms*]
by (*metis ground-term-to-gtermD*)

lemma *ta-reachableI* [*intro*]:
assumes *ground t* **and** *q* |∈| *ta-der* $\mathcal{A}$ *t*
shows *q* |∈| *ta-reachable* $\mathcal{A}$
using *assms finite-ta-reachable*
by (*auto simp*: *ta-reachable-def*)

lemma *ta-reachable-gtermI* [*intro*]:
q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟹ *q* |∈| *ta-reachable* $\mathcal{A}$

by (*intro ta-reachableI*[*of term-of-gterm t*]) *simp*

lemma *ta-reachableI-rule*:
assumes *sub*: *fset-of-list qs* |⊆| *ta-reachable* $\mathcal{A}$
and *rule*: *TA-rule f qs q* |∈| *rules* $\mathcal{A}$
shows *q* |∈| *ta-reachable* $\mathcal{A}$
∃ *ts*. *length qs* = *length ts* ∧ (∀ *i* < *length ts*. *ground* (*ts* ! *i*)) ∧
(∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*)) (**is** *?G*)
proof −
{
fix *i*
assume *i*: *i* < *length qs*
then have *qs* ! *i* |∈| *fset-of-list qs* **by** *auto*
with *sub* **have** *qs* ! *i* |∈| *ta-reachable* $\mathcal{A}$ **by** *auto*
from *ta-reachableE*[*OF this*] **have** ∃ *t*. *ground t* ∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *t* **by** *auto*
}
then have ∀ *i*. ∃ *t*. *i* < *length qs* ⟶ *ground t* ∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *t* **by** *auto*
from *choice*[*OF this*] **obtain** *ts* **where** *ts*: ⋀ *i*. *i* < *length qs* ⟹ *ground* (*ts i*)
∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts i*) **by** *blast*
let *?t* = *Fun f* (*map ts* [*0* ..< *length qs*])
have *gt*: *ground ?t* **using** *ts* **by** *auto*
have *r*: *q* |∈| *ta-der* $\mathcal{A}$ *?t* **unfolding** *ta-der-Fun* **using** *rule ts*
by (*intro exI*[*of* - *qs*] *exI*[*of* - *q*]) *simp*
with *gt* **show** *q* |∈| *ta-reachable* $\mathcal{A}$ **by** *blast*
from *gt ts* **show** *?G* **by** (*intro exI*[*of* - *map ts* [*0*..<*length qs*]]) *simp*
qed

lemma *ta-reachable-rule-gtermE*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
and *TA-rule f qs q* |∈| *rules* $\mathcal{A}$
obtains *t* **where** *groot t* = (*f*, *length qs*) *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
proof −
assume ∗: ⋀*t*. *groot t* = (*f*, *length qs*) ⟹ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟹
thesis
from *assms* **have** *fset-of-list qs* |⊆| *ta-reachable* $\mathcal{A}$
by (*auto dest*: *rule-statesD*(*3*))
from *ta-reachableI-rule*[*OF this assms*(*2*)] **obtain** *ts* **where** *args*: *length qs* =
length ts
∀ *i* < *length ts*. *ground* (*ts* ! *i*) ∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*)
using *assms* **by** *force*
then show *?thesis* **using** *assms*(*2*)
by (*intro* ∗[*of GFun f* (*map gterm-of-term ts*)]) *auto*
qed

lemma *ta-reachableI-eps′*:
assumes *reach*: *q* |∈| *ta-reachable* $\mathcal{A}$
and *eps*: (*q*, *q′*) |∈| (*eps* $\mathcal{A}$)|$^+$|
shows *q′* |∈| *ta-reachable* $\mathcal{A}$
proof −

from *ta-reachableE*[*OF reach*] **obtain** t **where** *g*: *ground* t **and** *res*: $q \mid\in\mid$ *ta-der* $\mathcal{A}$ t **by** *auto*
from *ta-der-trancl-eps*[*OF eps res*] *g* **show** *?thesis* **by** *blast*
qed

lemma *ta-reachableI-eps*:
assumes *reach*: $q \mid\in\mid$ *ta-reachable* $\mathcal{A}$
and *eps*: $(q, q') \mid\in\mid$ *eps* $\mathcal{A}$
shows $q' \mid\in\mid$ *ta-reachable* $\mathcal{A}$
by (*rule ta-reachableI-eps'*[*OF reach*], *insert eps*, *auto*)

— Automata are productive on a set P if all states can reach a state in P

lemma *finite-ta-productive*:
finite $\{p.\ \exists\, q\ q'\ C.\ p = q \wedge q' \mid\in\mid$ *ta-der* $\mathcal{A}$ $C\langle Var\ q\rangle \wedge q' \mid\in\mid P\}$
proof −
{fix x q C **assume** *ass*: $x \notin$ *fset* P $q \mid\in\mid P$ $q \mid\in\mid$ *ta-der* $\mathcal{A}$ $C\langle Var\ x\rangle$
then have $x \in$ *fset* $(\mathcal{Q}\ \mathcal{A})$
proof (*cases is-Fun* $C\langle Var\ x\rangle$)
case *True*
then show *?thesis* **using** *ta-der-is-fun-fvars-stateD*[*OF - ass*(*3*)]
by *auto*
next
case *False*
then show *?thesis* **using** *ass*
by (*cases* C, *auto*, (*metis eps-trancl-statesD*)+)
qed}
then have $\{q \mid q\ q'\ C.\ q' \mid\in\mid$ *ta-der* $\mathcal{A}$ $(C\langle Var\ q\rangle) \wedge q' \mid\in\mid P\} \subseteq$ *fset* $(\mathcal{Q}\ \mathcal{A}) \cup$ *fset* P **by** *auto*
from *finite-subset*[*OF this*] **show** *?thesis* **by** *auto*
qed

lemma *ta-productiveE*: **assumes** $q \mid\in\mid$ *ta-productive* P $\mathcal{A}$
obtains q' C **where** $q' \mid\in\mid$ *ta-der* $\mathcal{A}$ $(C\langle Var\ q\rangle)$ $q' \mid\in\mid P$
using *assms*[*unfolded ta-productive-def*] **by** *auto*

lemma *ta-productiveI*:
assumes $q' \mid\in\mid$ *ta-der* $\mathcal{A}$ $(C\langle Var\ q\rangle)$ $q' \mid\in\mid P$
shows $q \mid\in\mid$ *ta-productive* P $\mathcal{A}$
using *assms* **unfolding** *ta-productive-def*
using *finite-ta-productive*
by *auto*

lemma *ta-productiveI'*:
assumes $q \mid\in\mid$ *ta-der* $\mathcal{A}$ $(C\langle Var\ p\rangle)$ $q \mid\in\mid$ *ta-productive* P $\mathcal{A}$
shows $p \mid\in\mid$ *ta-productive* P $\mathcal{A}$
using *assms* **unfolding** *ta-productive-def*
by *auto* (*metis* (*mono-tags*, *lifting*) *ctxt-ctxt-compose ta-der-ctxt*)

lemma *ta-productive-setI*:
q |∈| *P* ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$
using *ta-productiveI*[*of q* $\mathcal{A}$ □ *q*]
by *simp*

lemma *ta-reachable-empty-rules* [*simp*]:
rules $\mathcal{A}$ = {||} ⟹ *ta-reachable* $\mathcal{A}$ = {||}
by (*auto simp*: *ta-reachable-def*)
(*metis ground.simps*(*1*) *ta.exhaust-sel ta-der-rule-empty*)

lemma *ta-reachable-mono*:
ta-subset $\mathcal{A}$ $\mathcal{B}$ ⟹ *ta-reachable* $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{B}$ **using** *ta-der-mono′*
by (*auto simp*: *ta-reachable-def*) *blast*

lemma *ta-reachabe-rhs-states*:
ta-reachable $\mathcal{A}$ |⊆| *ta-rhs-states* $\mathcal{A}$
proof −
{fix *q* **assume** *q* |∈| *ta-reachable* $\mathcal{A}$
then obtain *t* **where** *ground t q* |∈| *ta-der* $\mathcal{A}$ *t*
by (*auto simp*: *ta-reachable-def*)
then have *q* |∈| *ta-rhs-states* $\mathcal{A}$
by (*cases t*) (*auto simp*: *ta-rhs-states-def intro*!: *bexI*)**}**
then show *?thesis* **by** *blast*
qed

lemma *ta-reachable-eps*:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *p* |∈| *ta-reachable* $\mathcal{A}$ ⟹ (*p*, *q*) |∈| (*fRestr* (*eps* $\mathcal{A}$) (*ta-reachable* $\mathcal{A}$))|$^+$|
proof (*induct rule*: *ftrancl-induct*)
case (*Base a b*)
then show *?case*
by (*metis fSigmaI finterI fr-into-trancl ta-reachableI-eps*)
next
case (*Step p q r*)
then have *q* |∈| *ta-reachable* $\mathcal{A}$ *r* |∈| *ta-reachable* $\mathcal{A}$
by (*metis ta-reachableI-eps ta-reachableI-eps′*)+
then show *?case* **using** *Step*
by (*metis fSigmaI finterI ftrancl-into-trancl*)
qed

lemma *ta-der-only-reach*:
assumes *fvars-term t* |⊆| *ta-reachable* $\mathcal{A}$
shows *ta-der* $\mathcal{A}$ *t* = *ta-der* (*ta-only-reach* $\mathcal{A}$) *t* (**is** *?LS* = *?RS*)
proof −
have *?RS* |⊆| *?LS* **using** *ta-der-mono′*[*OF ta-restrict-subset*]
by *fastforce*

```
  moreover
  {fix q assume q |∈| ?LS
    then have q |∈| ?RS using assms
    proof (induct rule: ta-der-induct)
      case (Fun f ts ps p q)
       from Fun(2, 6) have ta-reach [simp]: i < length ps ⟹ fvars-term (ts ! i)
|⊆| ta-reachable 𝒜 for i
        by auto (metis ffUnionI fimage-fset fnth-mem funionI2 length-map nth-map
sup.orderE)
       from Fun have r: i < length ts ⟹ ps ! i |∈| ta-der (ta-only-reach 𝒜) (ts !
i)
         i < length ts ⟹ ps ! i |∈| ta-reachable 𝒜 for i
         by (auto) (metis ta-reach ta-der-ta-derict-states)+
       then have f ps → p |∈| rules (ta-only-reach 𝒜)
         using Fun(1, 2)
         by (intro ta-derict-ruleI)
           (fastforce simp: in-fset-conv-nth intro!: ta-reachableI-rule[OF - Fun(1)])+
     then show ?case using ta-reachable-eps[of p q] ta-reachableI-rule[OF - Fun(1)]
r Fun(2, 3)
       by (auto simp: ta-restrict-def intro!: exI[of - p] exI[of - ps])
    qed (auto simp: ta-restrict-def intro: ta-reachable-eps)}
  ultimately show ?thesis by blast
qed

lemma ta-der-gterm-only-reach:
  ta-der 𝒜 (term-of-gterm t) = ta-der (ta-only-reach 𝒜) (term-of-gterm t)
  using ta-der-only-reach[of term-of-gterm t 𝒜]
  by simp

lemma ta-reachable-ta-only-reach [simp]:
  ta-reachable (ta-only-reach 𝒜) = ta-reachable 𝒜  (is ?LS = ?RS)
proof −
  have ?LS |⊆| ?RS using ta-der-mono'[OF ta-restrict-subset]
    by (auto simp: ta-reachable-def) fastforce
  moreover
  {fix t assume ground (t :: ('b, 'a) term)
    then have ta-der 𝒜 t = ta-der (ta-only-reach 𝒜) t using ta-der-only-reach[of
t 𝒜]
      by simp}
  ultimately show ?thesis unfolding ta-reachable-def
    by auto
qed

lemma ta-only-reach-reachable:
  𝒬 (ta-only-reach 𝒜) |⊆| ta-reachable (ta-only-reach 𝒜)
  using ta-restrict-states-Q[of 𝒜 ta-reachable 𝒜]
  by auto
```

lemma *gta-only-reach-lang*:
gta-lang Q (*ta-only-reach* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
using *ta-der-gterm-only-reach*
by (*auto elim*!: *gta-langE intro*!: *gta-langI*) *force*+

lemma $\mathcal{L}$*-only-reach*: $\mathcal{L}$ (*reg-reach R*) = $\mathcal{L}$ *R*
using *gta-only-reach-lang*
by (*auto simp*: $\mathcal{L}$*-def reg-reach-def*)

lemma *ta-only-reach-lang*:
ta-lang Q (*ta-only-reach* $\mathcal{A}$) = *ta-lang Q* $\mathcal{A}$
using *gta-only-reach-lang*
by (*metis ta-lang-to-gta-lang*)

lemma *ta-prod-epsD*:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ *p* |∈| *ta-productive P* $\mathcal{A}$
using *ta-der-ctxt*[*of q* $\mathcal{A}$ □⟨*Var p*⟩]
by (*auto simp*: *ta-productive-def ta-der-trancl-eps*)

lemma *ta-only-prod-eps*:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ (*p*, *q*) |∈| (*eps* (*ta-only-prod P* $\mathcal{A}$))|$^+$|
proof (*induct rule*: *ftrancl-induct*)
case (*Base p q*)
then show *?case*
by (*metis* (*no-types*, *lifting*) *fSigmaI finterI fr-into-trancl ta.sel*(*2*) *ta-prod-epsD ta-restrict-def*)
next
case (*Step p q r*) **note** *IS* = *this*
show *?case* **using** *IS*(*2* − *4*) *ta-prod-epsD*[*OF fr-into-trancl*[*OF IS*(*3*)] *IS*(*4*)]
by (*auto simp*: *ta-restrict-def*) (*simp add*: *ftrancl-into-trancl*)
qed

lemma *ta-der-only-prod*:
q |∈| *ta-der* $\mathcal{A}$ *t* ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ *q* |∈| *ta-der* (*ta-only-prod P* $\mathcal{A}$) *t*
proof (*induct rule*: *ta-der-induct*)
case (*Fun f ts ps p q*)
let *?*$\mathcal{A}$ = *ta-only-prod P* $\mathcal{A}$
have *pr*: *p* |∈| *ta-productive P* $\mathcal{A}$ *i* < *length ts* ⟹ *ps* ! *i* |∈| *ta-productive P* $\mathcal{A}$ **for** *i*
using *Fun*(*2*) *ta-prod-epsD*[*of p q*] *Fun*(*3*, *6*) *rule-reachable-ctxt-exist*[*OF Fun*(*1*)]
using *ta-productiveI'*[*of p* $\mathcal{A}$ *- ps* ! *i P*]
by *auto*
then have *f ps* → *p* |∈| *rules ?*$\mathcal{A}$ **using** *Fun*(*1*, *2*) **unfolding** *ta-restrict-def*
by (*auto simp*: *in-fset-conv-nth intro*: *finite-subset*[*OF - finite-Collect-ta-rule*,

of - $\mathcal{A}$])
then show *?case* **using** *pr Fun ta-only-prod-eps*[*of p q* $\mathcal{A}$ *P*] *Fun*(*3*, *6*)
by *auto*
qed (*auto intro*: *ta-only-prod-eps*)

lemma *ta-der-ta-only-prod-ta-der*:
q |∈| *ta-der* (*ta-only-prod P* $\mathcal{A}$) $t \Longrightarrow q$ |∈| *ta-der* $\mathcal{A}$ t
by (*meson ta-der-el-mono ta-restrict-subset ta-subset-def*)

lemma *gta-only-prod-lang*:
gta-lang Q (*ta-only-prod Q* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$ (**is** *gta-lang Q* ?$\mathcal{A}$ = -)
proof
show *gta-lang Q* ?$\mathcal{A}$ ⊆ *gta-lang Q* $\mathcal{A}$
using *gta-lang-mono*[*OF ta-der-mono'*[*OF ta-restrict-subset*]]
by *blast*
next
{fix t **assume** $t \in$ *gta-lang Q* $\mathcal{A}$
from *gta-langE*[*OF this*] **obtain** q **where**
reach: q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) q |∈| *Q* **.**
from *ta-der-only-prod*[*OF reach*(*1*) *ta-productive-setI*[*OF reach*(*2*)]] *reach*(*2*)
have $t \in$ *gta-lang Q* ?$\mathcal{A}$ **by** (*auto intro*: *gta-langI*)**}**
then show *gta-lang Q* $\mathcal{A}$ ⊆ *gta-lang Q* ?$\mathcal{A}$ **by** *blast*
qed

lemma $\mathcal{L}$*-only-prod*: $\mathcal{L}$ (*reg-prod R*) = $\mathcal{L}$ *R*
using *gta-only-prod-lang*
by (*auto simp*: $\mathcal{L}$*-def reg-prod-def*)

lemma *ta-only-prod-lang*:
ta-lang Q (*ta-only-prod Q* $\mathcal{A}$) = *ta-lang Q* $\mathcal{A}$
using *gta-only-prod-lang*
by (*metis ta-lang-to-gta-lang*)

lemma *ta-prodictive-ta-only-prod* [*simp*]:
ta-productive P (*ta-only-prod P* $\mathcal{A}$) = *ta-productive P* $\mathcal{A}$ (**is** *?LS* = *?RS*)
proof −
have *?LS* |⊆| *?RS* **using** *ta-der-mono'*[*OF ta-restrict-subset*]
using *finite-ta-productive*[*of* $\mathcal{A}$ *P*]
by (*auto simp*: *ta-productive-def*) *fastforce*
moreover have *?RS* |⊆| *?LS* **using** *ta-der-only-prod*
by (*auto elim*!: *ta-productiveE*)
(*smt* (*verit*) *ta-der-only-prod ta-productiveI ta-productive-setI*)
ultimately show *?thesis* **by** *blast*
qed

lemma *ta-only-prod-productive*:

$\mathcal{Q}$ (*ta-only-prod P* $\mathcal{A}$) |⊆| *ta-productive P* (*ta-only-prod P* $\mathcal{A}$)
using *ta-restrict-states-Q* **by** *force*

lemma *ta-only-prod-reachable*:
assumes *all-reach*: $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows $\mathcal{Q}$ (*ta-only-prod P* $\mathcal{A}$) |⊆| *ta-reachable* (*ta-only-prod P* $\mathcal{A}$) (**is** *?Ls* |⊆| *?Rs*)
proof −
{fix *q* **assume** *q* |∈| *?Ls*
then obtain *t* **where** *ground t q* |∈| *ta-der* $\mathcal{A}$ *t q* |∈| *ta-productive P* $\mathcal{A}$
using *fsubsetD*[*OF ta-only-prod-productive*[*of* $\mathcal{A}$ *P*]]
using *fsubsetD*[*OF fsubset-trans*[*OF ta-restrict-states all-reach*, *of ta-productive P* $\mathcal{A}$]]
by (*auto elim*!: *ta-reachableE*)
then have *q* |∈| *?Rs*
by (*intro ta-reachableI*[**where** *?*$\mathcal{A}$ = *ta-only-prod P* $\mathcal{A}$ **and** *?t* = *t*]) (*auto simp*: *ta-der-only-prod*)**}**
then show *?thesis* **by** *blast*
qed

lemma *ta-prod-reach-subset*:
ta-subset (*ta-only-prod P* (*ta-only-reach* $\mathcal{A}$)) $\mathcal{A}$
by (*rule ta-subset-trans*, (*rule ta-restrict-subset*)+)

lemma *ta-prod-reach-states*:
$\mathcal{Q}$ (*ta-only-prod P* (*ta-only-reach* $\mathcal{A}$)) |⊆| $\mathcal{Q}$ $\mathcal{A}$
by (*rule ta-subset-states*[*OF ta-prod-reach-subset*])

lemma *ta-productive-aux*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$ *q* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*t*⟩)
shows ∃ *C'*. *ground-ctxt C'* ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*C'*⟨*t*⟩) **using** *assms*(*2*)
proof (*induct C arbitrary*: *q*)
case *Hole* **then show** *?case* **by** (*intro exI*[*of* - □]) *auto*
next
case (*More f ts1 C ts2*)
from *More*(*2*) **obtain** *qs q'* **where** *q'*: *f qs* → *q'* |∈| *rules* $\mathcal{A}$ *q'* = *q* ∨ (*q'*, *q*) |∈| (*eps* $\mathcal{A}$)|^+^|
qs ! *length ts1* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*t*⟩) *length qs* = *Suc* (*length ts1* + *length ts2*)
by *simp* (*metis less-add-Suc1 nth-append-length*)
{ fix *i* **assume** *i* < *length qs*
then have *qs* ! *i* |∈| $\mathcal{Q}$ $\mathcal{A}$ **using** *q'*(*1*)
by (*auto dest*!: *rule-statesD*(*4*))
then have ∃ *t*. *ground t* ∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *t* **using** *assms*(*1*)
by (*simp add*: *ta-reachable-def*) *force***}**
then obtain *ts* **where** *ts*: *i* < *length qs* ⟹ *ground* (*ts i*) ∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts i*) **for** *i* **by** *metis*
obtain *C'* **where** *C*: *ground-ctxt C' qs* ! *length ts1* |∈| *ta-der* $\mathcal{A}$ *C'*⟨*t*⟩ **using** *More*(*1*)[*OF q'*(*3*)] **by** *blast*
define *D* **where** *D* ≡ *More f* (*map ts* [*0*..<*length ts1*]) *C'* (*map ts* [*Suc* (*length*

ts1)..<*Suc* (*length ts1* + *length ts2*)])
have *ground-ctxt D* **unfolding** *D-def* **using** *ts C*(*1*) *q'*(*4*) **by** *auto*
moreover have *q* |∈| *ta-der* $\mathcal{A}$ *D*⟨*t*⟩ **using** *ts C*(*2*) *q'* **unfolding** *D-def*
by (*auto simp*: *nth-append-Cons not-le not-less le-less-Suc-eq Suc-le-eq intro*!: *exI*[*of* - *qs*] *exI*[*of* - *q'*])
ultimately show *?case* **by** *blast*
qed

lemma *ta-productive-def'*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows *ta-productive Q* $\mathcal{A}$ = {| *q*| *q q' C*. *ground-ctxt C* ∧ *q'* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*Var q*⟩) ∧ *q'* |∈| *Q* |}
using *ta-productive-aux*[*OF assms*]
by (*auto simp*: *ta-productive-def intro*!: *finite-subset*[*OF* - *finite-ta-productive*, *of* - $\mathcal{A}$ *Q*]) *force*+

lemma *trim-gta-lang*: *gta-lang Q* (*trim-ta Q* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
unfolding *trim-ta-def gta-only-reach-lang gta-only-prod-lang* **..**

lemma *trim-ta-subset*: *ta-subset* (*trim-ta Q* $\mathcal{A}$) $\mathcal{A}$
unfolding *trim-ta-def* **by** (*rule ta-prod-reach-subset*)

theorem *trim-ta*: *ta-is-trim Q* (*trim-ta Q* $\mathcal{A}$) **unfolding** *ta-is-trim-def*
by (*metis fin-mono ta-only-prod-reachable ta-only-reach-reachable*
ta-prodictive-ta-only-prod ta-restrict-states-Q trim-ta-def)

lemma *reg-is-trim-trim-reg* [*simp*]: *reg-is-trim* (*trim-reg R*)
unfolding *reg-is-trim-def trim-reg-def*
by (*simp add*: *trim-ta*)

lemma *trim-reg-reach* [*simp*]:
$\mathcal{Q}_r$ (*trim-reg A*) |⊆| *ta-reachable* (*ta* (*trim-reg A*))
by (*auto simp*: *trim-reg-def*) (*meson ta-is-trim-def trim-ta*)

lemma *trim-reg-prod* [*simp*]:
$\mathcal{Q}_r$ (*trim-reg A*) |⊆| *ta-productive* (*fin* (*trim-reg A*)) (*ta* (*trim-reg A*))
by (*auto simp*: *trim-reg-def*) (*meson ta-is-trim-def trim-ta*)

lemmas *obtain-trimmed-ta* = *trim-ta trim-gta-lang ta-subset-det*[*OF trim-ta-subset*]

lemma $\mathcal{L}$-*trim-ta-sig*:
assumes *reg-is-trim R* $\mathcal{L}$ *R* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
shows *ta-sig* (*ta R*) |⊆| $\mathcal{F}$
proof −

{fix r **assume** r: $r \mid\in\mid$ *rules* (*ta* R)
then obtain f *ps* p **where** [*simp*]: $r = f\ ps \to p$ **by** (*cases* r) *auto*
from r *assms*(*1*) **have** *fset-of-list ps* $\mid\subseteq\mid$ *ta-reachable* (*ta* R)
by (*auto simp add*: *rule-statesD*(*4*) *reg-is-trim-def ta-is-trim-def*)
from *ta-reachableI-rule*[*OF this*, *of f p*] r
obtain *ts* **where** *ts*: *length ts* = *length ps* $\forall\ i <$ *length ps*. *ground* (*ts* ! *i*)
$\forall\ i <$ *length ps*. *ps* ! *i* $\mid\in\mid$ *ta-der* (*ta* R) (*ts* ! *i*)
by *auto*
obtain C q **where** *ctxt*: *ground-ctxt* C q $\mid\in\mid$ *ta-der* (*ta* R) ($C\langle$*Var p*$\rangle$) q $\mid\in\mid$ *fin* R
using *assms*(*1*) **unfolding** *reg-is-trim-def*
by (*metis* ‹$r = f\ ps \to p$› *fsubsetI r rule-statesD*(*2*) *ta-productiveE ta-productive-aux ta-is-trim-def*)
from *ts ctxt r* **have** *reach*: $q \mid\in\mid$ *ta-der* (*ta* R) $C\langle$*Fun f ts*$\rangle$
by *auto* (*metis ta-der-Fun ta-der-ctxt*)
have *gr*: *ground* $C\langle$*Fun f ts*$\rangle$ **using** *ts*(*1*, *2*) *ctxt*(*1*)
by (*auto simp*: *in-set-conv-nth*)
then have $C\langle$*Fun f ts*$\rangle \in$ *ta-lang* (*fin* R) (*ta* R) **using** *ctxt*(*1*, *3*) *ts*(*1*, *2*)
apply (*intro ta-langI*[*OF - - reach*, *of fin R* $C\langle$*Fun f ts*$\rangle$])
apply (*auto simp del*: *adapt-vars-ctxt*)
by (*metis gr adapt-vars2 adapt-vars-adapt-vars*)
then have $*$: *gterm-of-term* $C\langle$*Fun f ts*$\rangle \in \mathcal{L}\ R$ **using** *gr*
by (*auto simp*: $\mathcal{L}$*-def*)
then have *funas-gterm* (*gterm-of-term* $C\langle$*Fun f ts*$\rangle$) $\subseteq$ *fset* $\mathcal{F}$ **using** *assms*(*2*) *gr*
by (*auto simp*: $\mathcal{T}_G$*-equivalent-def*)
moreover have (*f*, *length ps*) $\in$ *funas-gterm* (*gterm-of-term* $C\langle$*Fun f ts*$\rangle$)
using *ts*(*1*) **by** (*auto simp*: *funas-gterm-gterm-of-term*[*OF gr*])
ultimately have (*r-root r*, *length* (*r-lhs-states r*)) $\mid\in\mid$ $\mathcal{F}$
by *auto***}**
then show *?thesis*
by (*auto simp*: *ta-sig-def*)
qed

Map function over TA rules which change states/signature

lemma *map-ta-rule-iff*:
map-ta-rule f g |‘| Δ = {|*TA-rule* (*g h*) (*map f qs*) (*f q*) | *h qs q*. *TA-rule h qs q* $\mid\in\mid$ Δ|}
apply (*intro fequalityI fsubsetI*)
apply (*auto simp add*: *rev-image-eqI*)
apply (*metis map-ta-rule-cases ta-rule.collapse*)
done

lemma $\mathcal{L}$*-trim*: $\mathcal{L}$ (*trim-reg* R) = $\mathcal{L}\ R$
by (*auto simp*: *trim-gta-lang* $\mathcal{L}$*-def trim-reg-def*)

lemma *fmap-funs-ta-def′*:
fmap-funs-ta h $\mathcal{A}$ = *TA* {|(*h f*) *qs* $\to$ *q* |*f qs q*. *f qs* $\to$ *q* $\mid\in\mid$ *rules* $\mathcal{A}$|} (*eps* $\mathcal{A}$)

unfolding *fmap-funs-ta-def map-ta-rule-iff* **by** *auto*

lemma *fmap-states-ta-def'*:
fmap-states-ta h $\mathcal{A}$ = *TA* {|*f* (*map h qs*) → *h q* |*f qs q*. *f qs* → *q* |∈| *rules* $\mathcal{A}$|}
(*map-both h* |‘| *eps* $\mathcal{A}$)
unfolding *fmap-states-ta-def map-ta-rule-iff* **by** *auto*

lemma *fmap-states* [*simp*]:
$\mathcal{Q}$ (*fmap-states-ta h* $\mathcal{A}$) = *h* |‘| $\mathcal{Q}$ $\mathcal{A}$
unfolding *fmap-states-ta-def* $\mathcal{Q}$*-def*
by *auto*

lemma *fmap-states-ta-sig* [*simp*]:
ta-sig (*fmap-states-ta f* $\mathcal{A}$) = *ta-sig* $\mathcal{A}$
by (*auto simp*: *fmap-states-ta-def ta-sig-def ta-rule.map-sel intro*: *fset.map-cong0*)

lemma *fmap-states-ta-eps-wit*:
assumes (*h p*, *q*) |∈| (*map-both h* |‘| *eps* $\mathcal{A}$)|$^+$| *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$) *p* |∈| $\mathcal{Q}$ $\mathcal{A}$
obtains *q'* **where** *q* = *h q'* (*p*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$| *q'* |∈| $\mathcal{Q}$ $\mathcal{A}$
using *assms*(*1*)[*unfolded ftrancl-map-both-fsubset*[*OF assms*(*2*), *of eps* $\mathcal{A}$, *simplified*]]
using ‹*finj-on h* ($\mathcal{Q}$ $\mathcal{A}$)›[*unfolded finj-on-def'*, *rule-format*, *OF* ‹*p* |∈| $\mathcal{Q}$ $\mathcal{A}$›]
by (*metis Pair-inject eps-trancl-statesD prod-fun-fimageE*)

lemma *ta-der-fmap-states-inv-superset*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| $\mathcal{B}$ *finj-on h* $\mathcal{B}$
and *q* |∈| *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*)
shows *the-finv-into* $\mathcal{B}$ *h q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **using** *assms*(*3*)
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
from *assms*(*1*, *2*) **have** *inj*: *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$) **using** *fsubset-finj-on* **by** *blast*
have *x* |∈| $\mathcal{Q}$ $\mathcal{A}$ ⟹ *the-finv-into* ($\mathcal{Q}$ $\mathcal{A}$) *h* (*h x*) = *the-finv-into* $\mathcal{B}$ *h* (*h x*) **for** *x*
using *assms*(*1*, *2*) **by** (*metis fsubsetD inj the-finv-into-f-f*)
then show *?case* **using** *GFun the-finv-into-f-f*[*OF inj*] *assms*(*1*)
by (*auto simp*: *fmap-states-ta-def' finj-on-def' rule-statesD eps-statesD*
elim!: *fmap-states-ta-eps-wit*[*OF - inj*]
intro!: *exI*[*of - the-finv-into* $\mathcal{B}$ *h p*])
qed

lemma *ta-der-fmap-states-inv*:
assumes *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$) *q* |∈| *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*)
shows *the-finv-into* ($\mathcal{Q}$ $\mathcal{A}$) *h q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
by (*simp add*: *ta-der-fmap-states-inv-superset assms*)

lemma *ta-der-to-fmap-states-der*:
assumes *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
shows *h q* |∈| *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*) **using** *assms*
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)

then show *?case*
using *ftrancl-map-prod-mono*[*of h eps* $\mathcal{A}$]
by (*auto simp*: *fmap-states-ta-def' intro*!: *exI*[*of - h p*] *exI*[*of - map h ps*])
qed

lemma *ta-der-fmap-states-conv*:
assumes *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$)
shows *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*) = *h* |‘| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
using *ta-der-to-fmap-states-der*[*of - $\mathcal{A}$ t*] *ta-der-fmap-states-inv*[*OF assms*]
using *f-the-finv-into-f*[*OF assms*] *finj-on-the-finv-into*[*OF assms*]
using *gterm-ta-der-states*
by (*auto intro*!: *rev-fimage-eqI*) *fastforce*

lemma *fmap-states-ta-det*:
assumes *finj-on f* ($\mathcal{Q}$ $\mathcal{A}$)
shows *ta-det* (*fmap-states-ta f* $\mathcal{A}$) = *ta-det* $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
proof
{fix *g ps p q* **assume** *ass*: *?Ls TA-rule g ps p* |∈| *rules* $\mathcal{A}$ *TA-rule g ps q* |∈| *rules* $\mathcal{A}$
then have *TA-rule g* (*map f ps*) (*f p*) |∈| *rules* (*fmap-states-ta f* $\mathcal{A}$)
TA-rule g (*map f ps*) (*f q*) |∈| *rules* (*fmap-states-ta f* $\mathcal{A}$)
by (*force simp*: *fmap-states-ta-def*)+
then have *p* = *q* **using** *ass finj-on-eq-iff*[*OF assms*]
by (*auto simp*: *ta-det-def*) (*meson rule-statesD*(*2*))**}**
then show *?Ls* ⟹ *?Rs*
by (*auto simp*: *ta-det-def fmap-states-ta-def'*)
next
{fix *g ps qs p q* **assume** *ass*: *?Rs TA-rule g ps p* |∈| *rules* $\mathcal{A}$ *TA-rule g qs q* |∈| *rules* $\mathcal{A}$
then have *map f ps* = *map f qs* ⟹ *ps* = *qs* **using** *finj-on-eq-iff*[*OF assms*]
by (*auto simp*: *map-eq-nth-conv in-fset-conv-nth dest*!: *rule-statesD*(*4*) *intro*!: *nth-equalityI*)**}**
then show *?Rs* ⟹ *?Ls* **using** *finj-on-eq-iff*[*OF assms*]
by (*auto simp*: *ta-det-def fmap-states-ta-def'*) *blast*
qed

lemma *fmap-states-ta-lang*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$) ⟹ *Q* |⊆| $\mathcal{Q}$ $\mathcal{A}$ ⟹ *gta-lang* (*f* |‘| *Q*) (*fmap-states-ta f* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
using *ta-der-fmap-states-conv*[*of f* $\mathcal{A}$]
by (*auto simp*: *finj-on-def' finj-on-eq-iff fsubsetD elim*!: *gta-langE intro*!: *gta-langI*)

lemma *fmap-states-ta-lang2*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$ |∪| *Q*) ⟹ *gta-lang* (*f* |‘| *Q*) (*fmap-states-ta f* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
using *ta-der-fmap-states-conv*[*OF fsubset-finj-on*[*of f* $\mathcal{Q}$ $\mathcal{A}$ |∪| *Q* $\mathcal{Q}$ $\mathcal{A}$]]
by (*auto simp*: *finj-on-def' elim*!: *gta-langE intro*!: *gta-langI*) *fastforce*

definition *funs-ta* :: ('*q*, '*f*) *ta* ⇒ '*f fset* **where**
funs-ta $\mathcal{A}$ = {|*f* |*f qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$|}

lemma *funs-ta*[*code*]:
funs-ta $\mathcal{A}$ = (λ *r*. *case r of TA-rule f ps p* ⇒ *f*) |‘| (*rules* $\mathcal{A}$) (**is** *?Ls* = *?Rs*)
by (*force simp*: *funs-ta-def rev-fimage-eqI simp flip*: *fset.set-map*
split!: *ta-rule.splits intro*!: *finite-subset*[*of* {*f*. ∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$} *fset ?Rs*])

lemma *finite-funs-ta* [*simp*]:
finite {*f*. ∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$}
by (*intro finite-subset*[*of* {*f*. ∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$} *fset* (*funs-ta* $\mathcal{A}$)])
(*auto simp*: *funs-ta rev-fimage-eqI simp flip*: *fset.set-map split*!: *ta-rule.splits*)

lemma *funs-taE* [*elim*]:
assumes *f* |∈| *funs-ta* $\mathcal{A}$
obtains *ps p* **where** *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ **using** *assms*
by (*auto simp*: *funs-ta-def*)

lemma *funs-taI* [*intro*]:
TA-rule f ps p |∈| *rules* $\mathcal{A}$ ⟹ *f* |∈| *funs-ta* $\mathcal{A}$
by (*auto simp*: *funs-ta-def*)

lemma *fmap-funs-ta-cong*:
(⋀*x*. *x* |∈| *funs-ta* $\mathcal{A}$ ⟹ *h x* = *k x*) ⟹ $\mathcal{A}$ = $\mathcal{B}$ ⟹ *fmap-funs-ta h* $\mathcal{A}$ = *fmap-funs-ta k* $\mathcal{B}$
by (*force simp*: *fmap-funs-ta-def'*)

lemma [*simp*]: {|*TA-rule f qs q* |*f qs q*. *TA-rule f qs q* |∈| *X*|} = *X*
by (*intro fset-eqI*; *case-tac x*) *auto*

lemma *fmap-funs-ta-id* [*simp*]:
fmap-funs-ta id $\mathcal{A}$ = $\mathcal{A}$ **by** (*simp add*: *fmap-funs-ta-def'*)

lemma *fmap-states-ta-id* [*simp*]:
fmap-states-ta id $\mathcal{A}$ = $\mathcal{A}$
by (*auto simp*: *fmap-states-ta-def map-ta-rule-iff prod.map-id0*)

lemmas *fmap-funs-ta-id'* [*simp*] = *fmap-funs-ta-id*[*unfolded id-def*]

lemma *fmap-funs-ta-comp*:
fmap-funs-ta h (*fmap-funs-ta k A*) = *fmap-funs-ta* (*h* ∘ *k*) *A*
proof −
have *r* |∈| *rules A* ⟹ *map-ta-rule id h* (*map-ta-rule id k r*) = *map-ta-rule id* (λ*x*. *h* (*k x*)) *r* **for** *r*
by (*cases r*) (*auto*)
then show *?thesis*
by (*force simp*: *fmap-funs-ta-def fimage-iff cong*: *fmap-funs-ta-cong*)
qed

lemma *fmap-funs-reg-comp*:
fmap-funs-reg h (*fmap-funs-reg k A*) = *fmap-funs-reg* (*h* ∘ *k*) *A*
using *fmap-funs-ta-comp* **unfolding** *fmap-funs-reg-def*
by *auto*

lemma *fmap-states-ta-comp*:
fmap-states-ta h (*fmap-states-ta k A*) = *fmap-states-ta* (*h* ∘ *k*) *A*
by (*auto simp*: *fmap-states-ta-def ta-rule.map-comp comp-def id-def prod.map-comp*)

lemma *funs-ta-fmap-funs-ta* [*simp*]:
funs-ta (*fmap-funs-ta f A*) = *f* |‘| *funs-ta A*
by (*auto simp*: *funs-ta fmap-funs-ta-def' comp-def fimage-iff*
split!: *ta-rule.splits*) *force*+

lemma *ta-der-funs-ta*:
q |∈| *ta-der A t* ⟹ *ffuns-term t* |⊆| *funs-ta A*
proof (*induct t arbitrary*: *q*)
case (*Fun f ts*)
then have *f* |∈| *funs-ta A* **by** (*auto simp*: *funs-ta-def*)
then show *?case* **using** *Fun*(*1*)[*OF nth-mem*, *THEN fsubsetD*] *Fun*(*2*)
by (*auto simp*: *in-fset-conv-nth*) *blast*+
qed *auto*

lemma *ta-der-fmap-funs-ta*:
q |∈| *ta-der A t* ⟹ *q* |∈| *ta-der* (*fmap-funs-ta f A*) (*map-funs-term f t*)
by (*induct t arbitrary*: *q*) (*auto 0 4 simp*: *fmap-funs-ta-def'*)

lemma *ta-der-fmap-states-ta*:
assumes *q* |∈| *ta-der A t*
shows *h q* |∈| *ta-der* (*fmap-states-ta h A*) (*map-vars-term h t*)
proof −
have [*intro*]: (*q*, *q'*) |∈| (*eps A*)|^+^| ⟹ (*h q*, *h q'*) |∈| (*eps* (*fmap-states-ta h A*))|^+^|
for *q q'*
by (*force intro*!: *ftrancl-map*[*of eps A*] *simp*: *fmap-states-ta-def*)
show *?thesis* **using** *assms*
proof (*induct rule*: *ta-der-induct*)
case (*Fun f ts ps p q*)
have *f* (*map h ps*) → *h p* |∈| *rules* (*fmap-states-ta h A*)
using *Fun*(*1*) **by** (*force simp*: *fmap-states-ta-def'*)
then show *?case* **using** *Fun* **by** (*auto 0 4*)
qed *auto*
qed

lemma *ta-der-fmap-states-ta-mono*:
shows *f* |‘| *ta-der A* (*term-of-gterm s*) |⊆| *ta-der* (*fmap-states-ta f A*) (*term-of-gterm s*)
using *ta-der-fmap-states-ta*[*of - A term-of-gterm s f*]
by (*simp add*: *fimage-fsubsetI ta-der-to-fmap-states-der*)

lemma *ta-der-fmap-states-ta-mono2*:
assumes *finj-on f* (*Q A*)
shows *ta-der* (*fmap-states-ta f A*) (*term-of-gterm s*) |⊆| *f* |‘| *ta-der A* (*term-of-gterm s*)
using *ta-der-fmap-states-conv*[*OF assms*] **by** *auto*

lemma *fmap-funs-ta-der′*:
q |∈| *ta-der* (*fmap-funs-ta h A*) *t* ⟹ ∃ *t′*. *q* |∈| *ta-der A t′* ∧ *map-funs-term h t′* = *t*
proof (*induct rule*: *ta-der-induct*)
case (*Var q v*)
then show *?case* **by** (*auto simp*: *fmap-funs-ta-def intro*!: *exI*[*of* - *Var v*])
next
case (*Fun f ts ps p q*)
obtain *f′ ts′* **where** *root*: *f* = *h f′ f′ ps* → *p* |∈| *rules A* **and**
⋀*i*. *i* < *length ts* ⟹ *ps* ! *i* |∈| *ta-der A* (*ts′ i*) ∧ *map-funs-term h* (*ts′ i*) = *ts* ! *i*
using *Fun*(*1*, *5*) **unfolding** *fmap-funs-ta-def′*
by *auto metis*
note [*simp*] = *conjunct1*[*OF this*(*3*)] *conjunct2*[*OF this*(*3*), *unfolded id-def*]
have [*simp*]: *p* = *q* ⟹ *f′ ps* → *q* |∈| *rules A* **using** *root*(*2*) **by** *auto*
show *?case* **using** *Fun*(*3*)
by (*auto simp*: *comp-def Fun root fmap-funs-ta-def′*
intro!: *exI*[*of* - *Fun f′* (*map ts′* [*0*..<*length ts*])] *exI*[*of* - *ps*] *exI*[*of* - *p*] *nth-equalityI*)
qed

lemma *fmap-funs-gta-lang*:
gta-lang Q (*fmap-funs-ta h A*) = *map-gterm h* ‘ *gta-lang Q A* (**is** *?Ls* = *?Rs*)
proof −
{fix *s* **assume** *s* ∈ *?Ls* **then obtain** *q* **where**
lang: *q* |∈| *Q q* |∈| *ta-der* (*fmap-funs-ta h A*) (*term-of-gterm s*)
by *auto*
from *fmap-funs-ta-der′*[*OF this*(*2*)] **obtain** *t* **where**
t: *q* |∈| *ta-der A t map-funs-term h t* = *term-of-gterm s ground t*
by (*metis ground-map-term ground-term-of-gterm*)
then have *s* ∈ *?Rs* **using** *map-gterm-of-term*[*OF t*(*3*), *of h id*] *lang*
by (*auto simp*: *gta-lang-def gta-der-def image-iff*)
(*metis fempty-iff finterI ground-term-to-gtermD map-term-of-gterm term-of-gterm-inv*)**}**
moreover have *?Rs* ⊆ *?Ls* **using** *ta-der-fmap-funs-ta*[*of* - *A* - *h*]
by (*auto elim*!: *gta-langE intro*!: *gta-langI*) *fastforce*
ultimately show *?thesis* **by** *blast*
qed

lemma *fmap-funs-*$\mathcal{L}$:
$\mathcal{L}$ (*fmap-funs-reg h R*) = *map-gterm h* ‘ $\mathcal{L}$ *R*
using *fmap-funs-gta-lang*[*of fin R h*]
by (*auto simp*: *fmap-funs-reg-def* $\mathcal{L}$*-def*)

lemma *ta-states-fmap-funs-ta* [*simp*]: $\mathcal{Q}$ (*fmap-funs-ta f A*) = $\mathcal{Q}$ *A*
by (*auto simp*: *fmap-funs-ta-def* $\mathcal{Q}$*-def*)

lemma *ta-reachable-fmap-funs-ta* [*simp*]:
ta-reachable (*fmap-funs-ta f A*) = *ta-reachable A* **unfolding** *ta-reachable-def*
by (*metis* (*mono-tags*, *lifting*) *fmap-funs-ta-der*′ *ta-der-fmap-funs-ta ground-map-term*)

lemma *fin-in-states*:
fin (*reg-Restr-*Q_f *R*) |⊆| $\mathcal{Q}_r$ (*reg-Restr-*Q_f *R*)
by (*auto simp*: *reg-Restr-*Q_f*-def*)

lemma *fmap-states-reg-Restr-*Q_f*-fin*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$) ⟹ *fin* (*fmap-states-reg f* (*reg-Restr-*Q_f *R*)) |⊆| $\mathcal{Q}_r$ (*fmap-states-reg f* (*reg-Restr-*Q_f *R*))
by (*auto simp*: *fmap-states-reg-def reg-Restr-*Q_f*-def*)

lemma $\mathcal{L}$*-fmap-states-reg-Inl-Inr* [*simp*]:
$\mathcal{L}$ (*fmap-states-reg Inl R*) = $\mathcal{L}$ *R*
$\mathcal{L}$ (*fmap-states-reg Inr R*) = $\mathcal{L}$ *R*
unfolding $\mathcal{L}$*-def fmap-states-reg-def*
by (*auto simp*: *finj-Inl-Inr intro*!: *fmap-states-ta-lang2*)

lemma *finite-Collect-prod-ta-rules*:
finite {*f qs* → (*a*, *b*) |*f qs a b*. *f map fst qs* → *a* |∈| *rules* $\mathcal{A}$ ∧ *f map snd qs* → *b* |∈| *rules* $\mathfrak{B}$} (**is** *finite ?set*)
proof −
have *?set* ⊆ (λ (*ra*, *rb*). *case ra of f ps* → *p* ⇒ *case rb of g qs* → *q* ⇒ *f* (*zip ps qs*) → (*p*, *q*)) ‘ (*srules* $\mathcal{A}$ × *srules* $\mathfrak{B}$)
by (*auto simp*: *srules-def image-iff split*!: *ta-rule.splits*)
(*metis ta-rule.inject zip-map-fst-snd*)
from *finite-imageI*[*of srules* $\mathcal{A}$ × *srules* $\mathfrak{B}$, *THEN finite-subset*[*OF this*]]
show *?thesis* **by** (*auto simp*: *srules-def*)
qed

— The product automaton of the automata A and B is constructed by applying the rules on pairs of states

lemmas *prod-eps-def* = *prod-epsLp-def prod-epsRp-def*

lemma *finite-prod-epsLp*:
finite (*Collect* (*prod-epsLp* $\mathcal{A}$ $\mathcal{B}$))
by (*intro finite-subset*[*of Collect* (*prod-epsLp* $\mathcal{A}$ $\mathcal{B}$) *fset* (($\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$) |×| $\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$)])
(*auto simp*: *prod-epsLp-def dest*: *eps-statesD*)

lemma *finite-prod-epsRp*:
finite (*Collect* (*prod-epsRp* $\mathcal{A}$ $\mathcal{B}$))

by (*intro finite-subset*[*of Collect* (*prod-epsRp* $\mathcal{A}$ $\mathcal{B}$) *fset* (($\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$) |×| $\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$)])
(*auto simp*: *prod-epsRp-def dest*: *eps-statesD*)
lemmas *finite-prod-eps* [*simp*] = *finite-prod-epsLp*[*unfolded prod-epsLp-def*] *finite-prod-epsRp*[*unfolded prod-epsRp-def*]

lemma [*simp*]: *f qs* → *q* |∈| *rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) ⟷ *f qs* → *q* |∈| *prod-ta-rules* $\mathcal{A}$ $\mathcal{B}$
r |∈| *rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) ⟷ *r* |∈| *prod-ta-rules* $\mathcal{A}$ $\mathcal{B}$
by (*auto simp*: *prod-ta-def*)

lemma *prod-ta-states*:
$\mathcal{Q}$ (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) |⊆| $\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$
proof −
{fix *q* **assume** *q* |∈| *rule-states* (*rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$))
then obtain *f ps p* **where** *f ps* → *p* |∈| *rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) **and** *q* |∈| *fset-of-list ps* ∨ *p* = *q*
by (*metis rule-statesE*)
then have *fst q* |∈| $\mathcal{Q}$ $\mathcal{A}$ ∧ *snd q* |∈| $\mathcal{Q}$ $\mathcal{B}$
using *rule-statesD*(*2*, *4*)[*of f map fst ps fst p* $\mathcal{A}$]
using *rule-statesD*(*2*, *4*)[*of f map snd ps snd p* $\mathcal{B}$]
by *auto***}**
moreover
{fix *q* **assume** *q* |∈| *eps-states* (*eps* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$)) **then have** *fst q* |∈| $\mathcal{Q}$ $\mathcal{A}$ ∧ *snd q* |∈| $\mathcal{Q}$ $\mathcal{B}$
by (*auto simp*: *eps-states-def prod-ta-def prod-eps-def dest*: *eps-statesD*)**}**
ultimately show *?thesis*
by (*auto simp*: $\mathcal{Q}$*-def*) *blast*+
qed

lemma *prod-ta-det*:
assumes *ta-det* $\mathcal{A}$ **and** *ta-det* $\mathcal{B}$
shows *ta-det* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$)
using *assms* **unfolding** *ta-det-def prod-ta-def prod-eps-def*
by *auto*

lemma *prod-ta-sig*:
ta-sig (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) |⊆| *ta-sig* $\mathcal{A}$ |∪| *ta-sig* $\mathcal{B}$
proof (*rule fsubsetI*)
fix *x*
assume *x* |∈| *ta-sig* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$)
hence *x* |∈| *ta-sig* $\mathcal{A}$ ∨ *x* |∈| *ta-sig* $\mathcal{B}$
unfolding *ta-sig-def prod-ta-def*
using *image-iff* **by** *fastforce*
thus *x* |∈| *ta-sig* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) ⟹ *x* |∈| *ta-sig* $\mathcal{A}$ |∪| *ta-sig* $\mathcal{B}$
by *simp*
qed

lemma *from-prod-eps*:

$(p,\ q)\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+| \Longrightarrow (snd\ p,\ snd\ q)\ |\notin|\ (eps\ \mathcal{B})|^+| \Longrightarrow snd\ p = snd\ q \wedge (fst\ p,\ fst\ q)\ |\in|\ (eps\ \mathcal{A})|^+|$
$(p,\ q)\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+| \Longrightarrow (fst\ p,\ fst\ q)\ |\notin|\ (eps\ \mathcal{A})|^+| \Longrightarrow fst\ p = fst\ q \wedge (snd\ p,\ snd\ q)\ |\in|\ (eps\ \mathcal{B})|^+|$
apply (*induct rule*: *ftrancl-induct*)
apply (*auto simp*: *prod-ta-def prod-eps-def intro*: *ftrancl-into-trancl*)
apply (*simp add*: *fr-into-trancl not-ftrancl-into*)+
done

lemma *to-prod-eps*$\mathcal{A}$:
$(p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+| \Longrightarrow r\ |\in|\ \mathcal{Q}\ \mathcal{B} \Longrightarrow ((p,\ r),\ (q,\ r))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
by (*induct rule*: *ftrancl-induct*)
(*auto simp*: *prod-ta-def prod-eps-def intro*: *fr-into-trancl ftrancl-into-trancl*)

lemma *to-prod-eps*$\mathcal{B}$:
$(p,\ q)\ |\in|\ (eps\ \mathcal{B})|^+| \Longrightarrow r\ |\in|\ \mathcal{Q}\ \mathcal{A} \Longrightarrow ((r,\ p),\ (r,\ q))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
by (*induct rule*: *ftrancl-induct*)
(*auto simp*: *prod-ta-def prod-eps-def intro*: *fr-into-trancl ftrancl-into-trancl*)

lemma *to-prod-eps*:
$(p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+| \Longrightarrow (p',\ q')\ |\in|\ (eps\ \mathcal{B})|^+| \Longrightarrow ((p,\ p'),\ (q,\ q'))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
proof (*induct rule*: *ftrancl-induct*)
case (*Base a b*)
show *?case* **using** *Base*(*2*, *1*)
proof (*induct rule*: *ftrancl-induct*)
case (*Base c d*)
then have $((a,\ c),\ b,\ c)\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$ **using** *finite-prod-eps*
by (*auto simp*: *prod-ta-def prod-eps-def dest*: *eps-statesD intro*!: *fr-into-trancl ftrancl-into-trancl*)
moreover have $((b,\ c),\ b,\ d)\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$ **using** *finite-prod-eps Base*
by (*auto simp*: *prod-ta-def prod-eps-def dest*: *eps-statesD intro*!: *fr-into-trancl ftrancl-into-trancl*)
ultimately show *?case*
by (*auto intro*: *ftrancl-trans*)
next
case (*Step p q r*)
then have $((b,\ q),\ b,\ r)\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$ **using** *finite-prod-eps*
by (*auto simp*: *prod-ta-def prod-eps-def dest*: *eps-statesD intro*!: *fr-into-trancl*)
then show *?case* **using** *Step*
by (*auto intro*: *ftrancl-trans*)
qed
next
case (*Step a b c*)
from *Step* **have** $q'\ |\in|\ \mathcal{Q}\ \mathcal{B}$
by (*auto dest*: *eps-trancl-statesD*)
then have $((b,\ q'),\ (c,q'))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
using *Step*(*3*) *finite-prod-eps*

by (*auto simp*: *prod-ta-def prod-eps-def intro*!: *fr-into-trancl*)
then show *?case* **using** *ftrancl-trans Step*
by *auto*
qed

lemma *prod-ta-der-to-$\mathcal{A}$-$\mathcal{B}$-der1*:
assumes q |∈| *ta-der* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) (*term-of-gterm* t)
shows *fst* q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* t) **using** *assms*
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
then show *?case*
by (*auto dest*: *from-prod-eps intro*!: *exI*[*of - map fst ps*] *exI*[*of - fst p*])
qed

lemma *prod-ta-der-to-$\mathcal{A}$-$\mathcal{B}$-der2*:
assumes q |∈| *ta-der* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) (*term-of-gterm* t)
shows *snd* q |∈| *ta-der* $\mathcal{B}$ (*term-of-gterm* t) **using** *assms*
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
then show *?case*
by (*auto dest*: *from-prod-eps intro*!: *exI*[*of - map snd ps*] *exI*[*of - snd p*])
qed

lemma *$\mathcal{A}$-$\mathcal{B}$-der-to-prod-ta*:
assumes *fst* q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* t) *snd* q |∈| *ta-der* $\mathcal{B}$ (*term-of-gterm* t)
shows q |∈| *ta-der* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) (*term-of-gterm* t) **using** *assms*
proof (*induct t arbitrary*: q)
case (*GFun f ts*)
from *GFun*(*2*, *3*) **obtain** *ps qs p q′* **where**
rules: *f ps* → *p* |∈| *rules* $\mathcal{A}$ *f qs* → *q′* |∈| *rules* $\mathcal{B}$ *length ps* = *length ts length ps* = *length qs* **and**
eps: *p* = *fst q* ∨ (*p*, *fst q*) |∈| (*eps* $\mathcal{A}$)|$^+$| *q′* = *snd q* ∨ (*q′*, *snd q*) |∈| (*eps* $\mathcal{B}$)|$^+$|
and
steps: ∀ *i* < *length qs*. *ps* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ts* ! *i*))
∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{B}$ (*term-of-gterm* (*ts* ! *i*))
by *auto*
from *rules* **have** *st*: *p* |∈| $\mathcal{Q}$ $\mathcal{A}$ *q′* |∈| $\mathcal{Q}$ $\mathcal{B}$ **by** (*auto dest*: *rule-statesD*)
have (*p*, *snd q*) = *q* ∨ ((*p*, *q′*), *q*) |∈| (*eps* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$))|$^+$| **using** *eps st*
using *to-prod-eps$\mathcal{B}$*[*of q′ snd q* $\mathcal{B}$ *fst q* $\mathcal{A}$]
using *to-prod-eps$\mathcal{A}$*[*of p fst q* $\mathcal{A}$ *snd q* $\mathcal{B}$]
using *to-prod-eps*[*of p fst q* $\mathcal{A}$ *q′ snd q* $\mathcal{B}$]
by (*cases p* = *fst q*; *cases q′* = *snd q*) (*auto simp*: *prod-ta-def*)
then show *?case* **using** *rules eps steps GFun*(*1*) *st*
by (*cases* (*p*, *snd q*) = *q*)
(*auto simp*: *finite-Collect-prod-ta-rules dest*: *to-prod-eps$\mathcal{B}$ intro*!: *exI*[*of - p*] *exI*[*of - q′*] *exI*[*of - zip ps qs*])
qed

lemma *prod-ta-der*:
q |∈| *ta-der* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) (*term-of-gterm t*) ⟷
fst q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ∧ *snd q* |∈| *ta-der* $\mathcal{B}$ (*term-of-gterm t*)
using *prod-ta-der-to-*$\mathcal{A}$*-*$\mathcal{B}$*-der1 prod-ta-der-to-*$\mathcal{A}$*-*$\mathcal{B}$*-der2* $\mathcal{A}$*-*$\mathcal{B}$*-der-to-prod-ta*
by *blast*

lemma *intersect-ta-gta-lang*:
gta-lang ($Q_{\mathcal{A}}$ |×| $Q_{\mathcal{B}}$) (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) = *gta-lang* $Q_{\mathcal{A}}$ $\mathcal{A}$ ∩ *gta-lang* $Q_{\mathcal{B}}$ $\mathcal{B}$
by (*auto simp*: *prod-ta-der elim*!: *gta-langE intro*: *gta-langI*)

lemma $\mathcal{L}$*-intersect*: $\mathcal{L}$ (*reg-intersect R L*) = $\mathcal{L}$ *R* ∩ $\mathcal{L}$ *L*
by (*auto simp*: *intersect-ta-gta-lang* $\mathcal{L}$*-def reg-intersect-def*)

lemma *intersect-ta-ta-lang*:
ta-lang ($Q_{\mathcal{A}}$ |×| $Q_{\mathcal{B}}$) (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) = *ta-lang* $Q_{\mathcal{A}}$ $\mathcal{A}$ ∩ *ta-lang* $Q_{\mathcal{B}}$ $\mathcal{B}$
using *intersect-ta-gta-lang*[*of* $Q_{\mathcal{A}}$ $Q_{\mathcal{B}}$ $\mathcal{A}$ $\mathcal{B}$]
by *auto* (*metis IntI imageI term-of-gterm-inv*)

— Union of tree automata

lemma *ta-union-ta-subset*:
ta-subset $\mathcal{A}$ (*ta-union* $\mathcal{A}$ $\mathcal{B}$) *ta-subset* $\mathcal{B}$ (*ta-union* $\mathcal{A}$ $\mathcal{B}$)
unfolding *ta-subset-def ta-union-def*
by *auto*

lemma *ta-union-states* [*simp*]:
$\mathcal{Q}$ (*ta-union* $\mathcal{A}$ $\mathcal{B}$) = $\mathcal{Q}$ $\mathcal{A}$ |∪| $\mathcal{Q}$ $\mathcal{B}$
by (*auto simp*: *ta-union-def* $\mathcal{Q}$*-def*)

lemma *ta-union-sig* [*simp*]:
ta-sig (*ta-union* $\mathcal{A}$ $\mathcal{B}$) = *ta-sig* $\mathcal{A}$ |∪| *ta-sig* $\mathcal{B}$
by (*auto simp*: *ta-union-def ta-sig-def*)

lemma *ta-union-eps-disj-states*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} **and** (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))|$^+$|
shows (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ∨ (*p*, *q*) |∈| (*eps* $\mathcal{B}$)|$^+$| **using** *assms*(*2*, *1*)
by (*induct rule*: *ftrancl-induct*)
(*auto simp*: *ta-union-def ftrancl-into-trancl dest*: *eps-statesD eps-trancl-statesD*)

lemma *eps-ta-union-eps* [*simp*]:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))|$^+$|
(*p*, *q*) |∈| (*eps* $\mathcal{B}$)|$^+$| ⟹ (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))|$^+$|
by (*auto simp add*: *in-ftrancl-UnI ta-union-def*)

lemma *disj-states-eps* [*simp*]:
$\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} ⟹ *f ps* → *p* |∈| *rules* $\mathcal{A}$ ⟹ (*p*, *q*) |∈| (*eps* $\mathcal{B}$)|$^+$| ⟷ *False*
$\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} ⟹ *f ps* → *p* |∈| *rules* $\mathcal{B}$ ⟹ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟷ *False*

by (*auto simp*: *rtrancl-eq-or-trancl dest*: *rule-statesD eps-trancl-statesD*)

lemma *ta-union-der-disj-states*:
assumes $\mathcal{Q}\ \mathcal{A}\ |\cap|\ \mathcal{Q}\ \mathcal{B} = \{||\}$ **and** $q\ |\in|$ *ta-der* (*ta-union* $\mathcal{A}\ \mathcal{B}$) t
shows $q\ |\in|$ *ta-der* $\mathcal{A}\ t \vee q\ |\in|$ *ta-der* $\mathcal{B}\ t$ **using** *assms*(*2*)
proof (*induct rule*: *ta-der-induct*)
case (*Var q v*)
then show *?case* **using** *ta-union-eps-disj-states*[*OF assms*(*1*)]
by *auto*
next
case (*Fun f ts ps p q*)
have *dist*: *fset-of-list ps* $|\subseteq|\ \mathcal{Q}\ \mathcal{A} \Longrightarrow i <$ *length ts* $\Longrightarrow$ *ps* ! *i* $|\in|$ *ta-der* $\mathcal{A}$ (*ts* ! *i*)
fset-of-list ps $|\subseteq|\ \mathcal{Q}\ \mathcal{B} \Longrightarrow i <$ *length ts* $\Longrightarrow$ *ps* ! *i* $|\in|$ *ta-der* $\mathcal{B}$ (*ts* ! *i*) **for** *i*
using *Fun*(*2*) *Fun*(*5*)[*of i*] *assms*(*1*)
by (*auto dest*!: *ta-der-not-stateD fsubsetD*)
from *Fun*(*1*) **consider** (*a*) *fset-of-list ps* $|\subseteq|\ \mathcal{Q}\ \mathcal{A}$ | (*b*) *fset-of-list ps* $|\subseteq|\ \mathcal{Q}\ \mathcal{B}$
by (*auto simp*: *ta-union-def dest*: *rule-statesD*)
then show *?case* **using** *dist Fun*(*1*, *2*) *assms*(*1*) *ta-union-eps-disj-states*[*OF assms*(*1*), *of p q*] *Fun*(*3*)
by (*cases*) (*auto simp*: *fsubsetI rule-statesD ta-union-def intro*!: *exI*[*of - p*] *exI*[*of - ps*])
qed

lemma *ta-union-der-disj-states′*:
assumes $\mathcal{Q}\ \mathcal{A}\ |\cap|\ \mathcal{Q}\ \mathcal{B} = \{||\}$
shows *ta-der* (*ta-union* $\mathcal{A}\ \mathcal{B}$) $t =$ *ta-der* $\mathcal{A}\ t\ |\cup|$ *ta-der* $\mathcal{B}\ t$
using *ta-union-der-disj-states*[*OF assms*] *ta-der-mono′ ta-union-ta-subset*
by (*auto*, *fastforce*) *blast*

lemma *ta-union-gta-lang*:
assumes $\mathcal{Q}\ \mathcal{A}\ |\cap|\ \mathcal{Q}\ \mathcal{B} = \{||\}$ **and** $Q_{\mathcal{A}}\ |\subseteq|\ \mathcal{Q}\ \mathcal{A}$ **and** $Q_{\mathcal{B}}\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$
shows*gta-lang* ($Q_{\mathcal{A}}\ |\cup|\ Q_{\mathcal{B}}$) (*ta-union* $\mathcal{A}\ \mathcal{B}$) = *gta-lang* $Q_{\mathcal{A}}\ \mathcal{A} \cup$ *gta-lang* $Q_{\mathcal{B}}\ \mathcal{B}$ (**is** *?Ls* = *?Rs*)
proof −
{fix *s* **assume** $s \in$ *?Ls* **then obtain** *q*
where *w*: $q\ |\in|\ Q_{\mathcal{A}}\ |\cup|\ Q_{\mathcal{B}}\ q\ |\in|$ *ta-der* (*ta-union* $\mathcal{A}\ \mathcal{B}$) (*term-of-gterm s*)
by (*auto elim*: *gta-langE*)
from *ta-union-der-disj-states*[*OF assms*(*1*) *w*(*2*)] **consider**
(*a*) $q\ |\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm s*) | $q\ |\in|$ *ta-der* $\mathcal{B}$ (*term-of-gterm s*) **by** *blast*
then have $s \in$ *?Rs* **using** *w*(*1*) *assms*
by (*cases*, *auto simp*: *gta-langI*)
(*metis fempty-iff finterI funion-iff gterm-ta-der-states sup.orderE*)**}**
moreover have *?Rs* $\subseteq$ *?Ls* **using** *ta-union-der-disj-states′*[*OF assms*(*1*)]
by (*auto elim*!: *gta-langE intro*!: *gta-langI*)
ultimately show *?thesis* **by** *blast*
qed

lemma $\mathcal{L}$*-union*: $\mathcal{L}$ (*reg-union R L*) = $\mathcal{L}$ *R* ∪ $\mathcal{L}$ *L*
proof −
let *?inl* = *Inl* :: *'b* ⇒ *'b* + *'c* **let** *?inr* = *Inr* :: *'c* ⇒ *'b* + *'c*
let *?fr* = *?inl* |`| (*fin R* |∩| $\mathcal{Q}_r$ *R*) **let** *?fl* = *?inr* |`| (*fin L* |∩| $\mathcal{Q}_r$ *L*)
have [*simp*]:*gta-lang* (*?fr* |∪| *?fl*) (*ta-union* (*fmap-states-ta ?inl* (*ta R*)) (*fmap-states-ta ?inr* (*ta L*))) =
gta-lang ?fr (*fmap-states-ta ?inl* (*ta R*)) ∪ *gta-lang ?fl* (*fmap-states-ta ?inr* (*ta L*))
by (*intro ta-union-gta-lang*) (*auto simp*: *fimage-iff*)
have [*simp*]: *gta-lang ?fr* (*fmap-states-ta ?inl* (*ta R*)) = *gta-lang* (*fin R* |∩| $\mathcal{Q}_r$ *R*) (*ta R*)
by (*intro fmap-states-ta-lang*) (*auto simp*: *finj-Inl-Inr*)
have [*simp*]: *gta-lang ?fl* (*fmap-states-ta ?inr* (*ta L*)) = *gta-lang* (*fin L* |∩| $\mathcal{Q}_r$ *L*) (*ta L*)
by (*intro fmap-states-ta-lang*) (*auto simp*: *finj-Inl-Inr*)
show *?thesis*
using *gta-lang-Rest-states-conv*
by (*auto simp*: $\mathcal{L}$*-def reg-union-def ta-union-gta-lang*) *fastforce*
qed

lemma *reg-union-states*:
$\mathcal{Q}_r$ (*reg-union A B*) = (*Inl* |`| $\mathcal{Q}_r$ *A*) |∪| (*Inr* |`| $\mathcal{Q}_r$ *B*)
by (*auto simp*: *reg-union-def*)

— Deciding emptiness

lemma *ta-empty* [*simp*]:
ta-empty Q $\mathcal{A}$ = (*gta-lang Q* $\mathcal{A}$ = {})
by (*auto simp*: *ta-empty-def elim*!: *gta-langE ta-reachable-gtermE*
intro: *ta-reachable-gtermI gta-langI*)

lemma *reg-empty* [*simp*]:
reg-empty R = ($\mathcal{L}$ *R* = {})
by (*simp add*: $\mathcal{L}$*-def reg-empty-def*)

Epsilon free automaton

lemma *finite-eps-free-rulep* [*simp*]:
finite (*Collect* (*eps-free-rulep* $\mathcal{A}$))
proof −
let *?par* = (λ *r*. *case r of f qs* → *q* ⇒ (*f*, *qs*)) |`| (*rules* $\mathcal{A}$)
let *?st* = (λ (*r*, *q*). *case r of* (*f*, *qs*) ⇒ *TA-rule f qs q*) |`| (*?par* |×| $\mathcal{Q}$ $\mathcal{A}$)
show *?thesis* **using** *rule-statesD eps-trancl-statesD*
by (*intro finite-subset*[*of Collect* (*eps-free-rulep* $\mathcal{A}$) *fset ?st*])
(*auto simp*: *eps-free-rulep-def fimage-iff*
simp flip: *fset.set-map*
split!: *ta-rule.splits*, *fastforce*+)
qed

lemmas *finite-eps-free-rule* [*simp*] = *finite-eps-free-rulep*[*unfolded eps-free-rulep-def*]

lemma *ta-res-eps-free*:
  ta-der (*eps-free* $\mathcal{A}$) (*term-of-gterm t*) = *ta-der* $\mathcal{A}$ (*term-of-gterm t*) (**is** *?Ls* = *?Rs*)
proof −
  {fix *q* **assume** *q* |∈| *?Ls* **then have** *q* |∈| *?Rs*
    by (*induct rule*: *ta-der-gterm-induct*)
      (*auto simp*: *eps-free-def eps-free-rulep-def*)**}**
  moreover
  {fix *q* **assume** *q* |∈| *?Rs* **then have** *q* |∈| *?Ls*
   proof (*induct rule*: *ta-der-gterm-induct*)
    case (*GFun f ts ps p q*)
    then show *?case*
     by (*auto simp*: *eps-free-def eps-free-rulep-def intro*!: *exI*[*of - ps*])
   qed}
  ultimately show *?thesis* **by** *blast*
qed

lemma *ta-lang-eps-free* [*simp*]:
  gta-lang Q (*eps-free* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
  by (*auto simp add*: *ta-res-eps-free elim*!: *gta-langE intro*: *gta-langI*)

lemma $\mathcal{L}$*-eps-free*: $\mathcal{L}$ (*eps-free-reg R*) = $\mathcal{L}$ *R*
  by (*auto simp*: $\mathcal{L}$*-def eps-free-reg-def*)

Sufficient criterion for containment

definition *ta-contains-aux* :: ('*f* × *nat*) *set* ⇒ '*q fset* ⇒ ('*q*, '*f*) *ta* ⇒ '*q fset* ⇒ *bool* **where**
  ta-contains-aux $\mathcal{F}$ Q_1 $\mathcal{A}$ Q_2 ≡ (∀ *f qs*. (*f*, *length qs*) ∈ $\mathcal{F}$ ∧ *fset-of-list qs* |⊆| Q_1 ⟶
    (∃ *q q'*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ *q'* |∈| Q_2 ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|))))

lemma *ta-contains-aux-state-set*:
  assumes *ta-contains-aux* $\mathcal{F}$ *Q* $\mathcal{A}$ *Q t* ∈ $\mathcal{T}_G$ $\mathcal{F}$
  shows ∃ *q*. *q* |∈| *Q* ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **using** *assms*(*2*)
proof (*induct rule*: $\mathcal{T}_G$*.induct*)
  case (*const a*)
  then show *?case* **using** *assms*(*1*)
   by (*force simp*: *ta-contains-aux-def*)
next
  case (*ind f n ss*)
  obtain *qs* **where** *fset-of-list qs* |⊆| *Q length ss* = *length qs*
   ∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ss* ! *i*))
    using *ind*(*4*) *Ex-list-of-length-P*[*of length ss* λ *q i*. *q* |∈| *Q* ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ss* ! *i*))]
   by (*auto simp*: *fset-list-fsubset-eq-nth-conv*) *metis*

then show *?case* **using** *ind*(*1* − *3*) *assms*(*1*)
by (*auto simp*: *ta-contains-aux-def*) *blast*
qed

lemma *ta-contains-aux-mono*:
assumes *ta-subset* $\mathcal{A}$ $\mathcal{B}$ **and** Q_2 |⊆| $Q_2{'}$
shows *ta-contains-aux* $\mathcal{F}$ Q_1 $\mathcal{A}$ Q_2 ⟹ *ta-contains-aux* $\mathcal{F}$ Q_1 $\mathcal{B}$ $Q_2{'}$
using *assms* **unfolding** *ta-contains-aux-def ta-subset-def*
by (*meson fin-mono ftrancl-mono*)

definition *ta-contains* :: ('*f* × *nat*) *set* ⇒ ('*f* × *nat*) *set* ⇒ ('*q*, '*f*) *ta* ⇒ '*q fset* ⇒ '*q fset* ⇒ *bool*
where *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{A}$ Q Q_f ≡ *ta-contains-aux* $\mathcal{F}$ Q $\mathcal{A}$ Q ∧ *ta-contains-aux* $\mathcal{G}$ Q $\mathcal{A}$ Q_f

lemma *ta-contains-mono*:
assumes *ta-subset* $\mathcal{A}$ $\mathcal{B}$ **and** Q_f |⊆| $Q_f{'}$
shows *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{A}$ Q Q_f ⟹ *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{B}$ Q $Q_f{'}$
unfolding *ta-contains-def*
using *ta-contains-aux-mono*[*OF assms*(*1*) *fsubset-refl*]
using *ta-contains-aux-mono*[*OF assms*]
by *blast*

lemma *ta-contains-both*:
assumes *contain*: *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{A}$ Q Q_f
shows ⋀ *t*. *groot t* ∈ $\mathcal{G}$ ⟹ ⋃ (*funas-gterm* ‘ *set* (*gargs t*)) ⊆ $\mathcal{F}$ ⟹ *t* ∈ *gta-lang* Q_f $\mathcal{A}$
proof −
fix *t* :: '*a gterm*
assume *F*: ⋃ (*funas-gterm* ‘ *set* (*gargs t*)) ⊆ $\mathcal{F}$ **and** *G*: *groot t* ∈ $\mathcal{G}$
obtain *g ss* **where** *t*[*simp*]: *t* = *GFun g ss* **by** (*cases t*, *auto*)
then have ∃ *qs*. *length qs* = *length ss* ∧ (∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ss* ! *i*)) ∧ *qs* ! *i* |∈| Q)
using *contain ta-contains-aux-state-set*[*of* $\mathcal{F}$ Q $\mathcal{A}$ *ss* ! *i* **for** *i*] *F*
unfolding *ta-contains-def* $\mathcal{T}_G$*-funas-gterm-conv*
using *Ex-list-of-length-P*[*of length ss* λ *q i*. *q* |∈| Q ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ss* ! *i*))]
by *auto* (*metis SUP-le-iff nth-mem*)
then obtain *qs* **where** *length qs* = *length ss*
∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ss* ! *i*))
∀ *i* < *length qs*. *qs* ! *i* |∈| Q
by *blast*
then obtain *q* **where** *q* |∈| Q_f *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
using *conjunct2*[*OF contain*[*unfolded ta-contains-def*]] *G*
by (*auto simp*: *ta-contains-def ta-contains-aux-def fset-list-fsubset-eq-nth-conv*) *metis*
then show *t* ∈ *gta-lang* Q_f $\mathcal{A}$
by (*intro gta-langI*) *simp*
qed

lemma *ta-contains*:
assumes *contain*: *ta-contains* $\mathcal{F}$ $\mathcal{F}$ $\mathcal{A}$ Q Q_f
shows $\mathcal{T}_G$ $\mathcal{F}$ $\subseteq$ *gta-lang* Q_f $\mathcal{A}$ (**is** *?A* $\subseteq$ -)
proof −
have [*simp*]: *funas-gterm* t $\subseteq$ $\mathcal{F}$ $\Longrightarrow$ *groot* t $\in$ $\mathcal{F}$ **for** t **by** (*cases* t) *auto*
have [*simp*]: *funas-gterm* t $\subseteq$ $\mathcal{F}$ $\Longrightarrow$ $\bigcup$ (*funas-gterm* ‘ *set* (*gargs* t)) $\subseteq$ $\mathcal{F}$ **for** t
by (*cases* t) *auto*
show *?thesis* **using** *ta-contains-both*[*OF contain*]
by (*auto simp*: $\mathcal{T}_G$-*equivalent-def*)
qed

Relabeling, map finite set to natural numbers

lemma *map-fset-to-nat-inj*:
assumes Y |$\subseteq$| X
shows *finj-on* (*map-fset-to-nat* X) Y
proof −
{ fix x y **assume** x |$\in$| X y |$\in$| X
then have x |$\in$| *fset-of-list* (*sorted-list-of-fset* X) y |$\in$| *fset-of-list* (*sorted-list-of-fset* X)
by *simp-all*
note *this*[*unfolded mem-idx-fset-sound*]
then have x = y **if** *map-fset-to-nat* X x = *map-fset-to-nat* X y
using *that nth-eq-iff-index-eq*[*OF distinct-sorted-list-of-fset*[*of* X]]
by (*force dest*: *mem-idx-sound-output simp*: *map-fset-to-nat-def*) **}**
then show *?thesis* **using** *assms*
by (*auto simp add*: *finj-on-def′ fBall-def*)
qed

lemma *map-fset-fset-to-nat-inj*:
assumes Y |$\subseteq$| X
shows *finj-on* (*map-fset-fset-to-nat* X) Y **using** *assms*
proof −
let *?f* = *map-fset-fset-to-nat* X
{ fix x y **assume** x |$\in$| X y |$\in$| X
then have *sorted-list-of-fset* x |$\in$| *fset-of-list* (*sorted-list-of-fset* (*sorted-list-of-fset* |‘| X))
sorted-list-of-fset y |$\in$| *fset-of-list* (*sorted-list-of-fset* (*sorted-list-of-fset* |‘| X))
unfolding *map-fset-fset-to-nat-def* **by** *auto*
note *this*[*unfolded mem-idx-fset-sound*]
then have x = y **if** *?f* x = *?f* y
using *that nth-eq-iff-index-eq*[*OF distinct-sorted-list-of-fset*[*of sorted-list-of-fset* |‘| X]]
by (*auto simp*: *map-fset-fset-to-nat-def*)
(*metis mem-idx-sound-output sorted-list-of-fset-simps*(*1*))+**}**
then show *?thesis* **using** *assms*
by (*auto simp add*: *finj-on-def′ fBall-def*)
qed

lemma *relabel-gta-lang* [*simp*]:
gta-lang (*relabel-Q_f* Q $\mathcal{A}$) (*relabel-ta* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$
proof −
have *gta-lang* (*relabel-Q_f* Q $\mathcal{A}$) (*relabel-ta* $\mathcal{A}$) = *gta-lang* (Q |∩| $\mathcal{Q}$ $\mathcal{A}$) $\mathcal{A}$
unfolding *relabel-ta-def relabel-Q_f-def*
by (*intro fmap-states-ta-lang2 map-fset-to-nat-inj*) *simp*
then show *?thesis* **by** *fastforce*
qed

lemma $\mathcal{L}$*-relable* [*simp*]: $\mathcal{L}$ (*relabel-reg* R) = $\mathcal{L}$ R
by (*auto simp*: $\mathcal{L}$*-def relabel-reg-def*)

lemma *relabel-ta-lang* [*simp*]:
ta-lang (*relabel-Q_f* Q $\mathcal{A}$) (*relabel-ta* $\mathcal{A}$) = *ta-lang* Q $\mathcal{A}$
unfolding *ta-lang-to-gta-lang*
using *relabel-gta-lang*
by *simp*

lemma *relabel-fset-gta-lang* [*simp*]:
gta-lang (*relabel-fset-Q_f* Q $\mathcal{A}$) (*relabel-fset-ta* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$
proof −
have *gta-lang* (*relabel-fset-Q_f* Q $\mathcal{A}$) (*relabel-fset-ta* $\mathcal{A}$) = *gta-lang* (Q |∩| $\mathcal{Q}$ $\mathcal{A}$) $\mathcal{A}$
unfolding *relabel-fset-Q_f-def relabel-fset-ta-def*
by (*intro fmap-states-ta-lang2 map-fset-fset-to-nat-inj*) *simp*
then show *?thesis* **by** *fastforce*
qed

lemma $\mathcal{L}$*-relable-fset* [*simp*]: $\mathcal{L}$ (*relable-fset-reg* R) = $\mathcal{L}$ R
by (*auto simp*: $\mathcal{L}$*-def relable-fset-reg-def*)

lemma *ta-states-trim-ta*:
$\mathcal{Q}$ (*trim-ta* Q $\mathcal{A}$) |⊆| $\mathcal{Q}$ $\mathcal{A}$
unfolding *trim-ta-def* **using** *ta-prod-reach-states* **.**

lemma *trim-ta-reach*: $\mathcal{Q}$ (*trim-ta* Q $\mathcal{A}$) |⊆| *ta-reachable* (*trim-ta* Q $\mathcal{A}$)
unfolding *trim-ta-def* **using** *ta-only-prod-reachable ta-only-reach-reachable*
by *metis*

lemma *trim-ta-prod*: $\mathcal{Q}$ (*trim-ta* Q A) |⊆| *ta-productive* Q (*trim-ta* Q A)
unfolding *trim-ta-def* **using** *ta-only-prod-productive*
by *metis*

lemma *empty-gta-lang*:

gta-lang Q (*TA* {||} {||}) = {}
using *ta-reachable-gtermI*
by (*force simp*: *gta-lang-def gta-der-def elim*!: *ta-langE*)

abbreviation *empty-reg* **where**
empty-reg ≡ *Reg* {||} (*TA* {||} {||})

lemma $\mathcal{L}$*-epmty*:
$\mathcal{L}$ *empty-reg* = {}
by (*auto simp*: $\mathcal{L}$*-def empty-gta-lang*)

lemma *const-ta-lang*:
gta-lang {|*q*|} (*TA* {| *TA-rule f* [] *q* |} {||}) = {*GFun f* []}
proof −
have [*dest*!]: *q*′ |∈| *ta-der* (*TA* {| *TA-rule f* [] *q* |} {||}) *t*′ ⟹ *ground t*′ ⟹ *t*′ = *Fun f* [] **for** *t*′ *q*′
by (*induct t*′) *auto*
show *?thesis*
by (*auto simp*: *gta-lang-def gta-der-def elim*!: *gta-langE*)
(*metis gterm-of-term.simps list.simps*(*8*) *term-of-gterm-inv*)
qed

lemma *run-argsD*:
run $\mathcal{A}$ *s t* ⟹ *length* (*gargs s*) = *length* (*gargs t*) ∧ (∀ *i* < *length* (*gargs t*). *run* $\mathcal{A}$ (*gargs s* ! *i*) (*gargs t* ! *i*))
using *run.cases* **by** *fastforce*

lemma *run-root-rule*:
run $\mathcal{A}$ *s t* ⟹ *TA-rule* (*groot-sym t*) (*map ex-comp-state* (*gargs s*)) (*ex-rule-state s*) |∈| (*rules* $\mathcal{A}$) ∧
(*ex-rule-state s* = *ex-comp-state s* ∨ (*ex-rule-state s*, *ex-comp-state s*) |∈| (*eps* $\mathcal{A}$)|$^+$|)
by (*cases s*; *cases t*) (*auto elim*: *run.cases*)

lemma *run-poss-eq*: *run* $\mathcal{A}$ *s t* ⟹ *gposs s* = *gposs t*
by (*induct rule*: *run.induct*) *auto*

lemma *run-gsubt-cl*:
assumes *run* $\mathcal{A}$ *s t* **and** *p* ∈ *gposs t*
shows *run* $\mathcal{A}$ (*gsubt-at s p*) (*gsubt-at t p*) **using** *assms*
proof (*induct p arbitrary*: *s t*)
case (*Cons i p*) **show** *?case* **using** *Cons*(*1*) *Cons*(*2*−)
by (*cases s*; *cases t*) (*auto dest*: *run-argsD*)
qed *auto*

lemma *run-replace-at*:
assumes *run* $\mathcal{A}$ *s t* **and** *run* $\mathcal{A}$ *u v* **and** *p* ∈ *gposs s*
and *ex-comp-state* (*gsubt-at s p*) = *ex-comp-state u*

shows *run* $\mathcal{A}$ $s[p \leftarrow u]_G$ $t[p \leftarrow v]_G$ **using** *assms*
proof (*induct p arbitrary*: *s t*)
case (*Cons i p*)
obtain *r q qs f ts* **where** [*simp*]: $s = GFun\ (r,\ q)\ qs\ t = GFun\ f\ ts$ **by** (*cases s*, *cases t*) *auto*
have $*$: $j < length\ qs \Longrightarrow ex\text{-}comp\text{-}state\ (qs[i := (qs\ !\ i)[p \leftarrow u]_G]\ !\ j) = ex\text{-}comp\text{-}state\ (qs\ !\ j)$ **for** *j*
using *Cons*(*5*) **by** (*cases i = j*, *cases p*) *auto*
have [*simp*]: *map ex-comp-state* $(qs[i := (qs\ !\ i)[p \leftarrow u]_G]) = map\ ex\text{-}comp\text{-}state\ qs$ **using** *Cons*(*5*)
by (*auto simp*: $*$[*unfolded comp-def*] *intro*!: *nth-equalityI*)
have *run* $\mathcal{A}$ $(qs\ !\ i)[p \leftarrow u]_G$ $(ts\ !\ i)[p \leftarrow v]_G$ **using** *Cons*(*2*−)
by (*intro Cons*(*1*)) (*auto dest*: *run-argsD*)
moreover have $i < length\ qs\ i < length\ ts$ **using** *Cons*(*4*) *run-poss-eq*[*OF Cons*(*2*)]
by *force*+
ultimately show *?case* **using** *Cons*(*2*) *run-root-rule*[*OF Cons*(*2*)]
by (*fastforce simp*: *nth-list-update dest*: *run-argsD intro*!: *run.intros*)
qed *simp*

relating runs to derivation definition

lemma *run-to-comp-st-gta-der*:
run $\mathcal{A}$ *s t* $\Longrightarrow$ *ex-comp-state s* $|\in|$ *gta-der* $\mathcal{A}$ *t*
proof (*induct s arbitrary*: *t*)
case (*GFun q qs*)
show *?case* **using** *GFun*(*1*)[*OF nth-mem*, *of i gargs t ! i* **for** *i*]
using *run-argsD*[*OF GFun*(*2*)] *run-root-rule*[*OF GFun*(*2*−)]
by (*cases t*) (*auto simp*: *gta-der-def intro*!: *exI*[*of - map ex-comp-state qs*] *exI*[*of - fst q*])
qed

lemma *run-to-rule-st-gta-der*:
assumes *run* $\mathcal{A}$ *s t* **shows** *ex-rule-state s* $|\in|$ *gta-der* $\mathcal{A}$ *t*
proof (*cases s*)
case [*simp*]: (*GFun q qs*)
have $i < length\ qs \Longrightarrow ex\text{-}comp\text{-}state\ (qs\ !\ i)\ |\in|\ gta\text{-}der\ \mathcal{A}\ (gargs\ t\ !\ i)$ **for** *i*
using *run-to-comp-st-gta-der*[*of* $\mathcal{A}$] *run-argsD*[*OF assms*] **by** *force*
then show *?thesis* **using** *conjunct1*[*OF run-argsD*[*OF assms*]] *run-root-rule*[*OF assms*]
by (*cases t*) (*auto simp*: *gta-der-def intro*!: *exI*[*of - map ex-comp-state qs*] *exI*[*of - fst q*])
qed

lemma *run-to-gta-der-gsubt-at*:
assumes *run* $\mathcal{A}$ *s t* **and** $p \in gposs\ t$
shows *ex-rule-state* $(gsubt\text{-}at\ s\ p)\ |\in|\ gta\text{-}der\ \mathcal{A}\ (gsubt\text{-}at\ t\ p)$
ex-comp-state $(gsubt\text{-}at\ s\ p)\ |\in|\ gta\text{-}der\ \mathcal{A}\ (gsubt\text{-}at\ t\ p)$
using *assms run-gsubt-cl*[*THEN run-to-comp-st-gta-der*] *run-gsubt-cl*[*THEN run-to-rule-st-gta-der*]
by *blast*+

lemma *gta-der-to-run*:
  q |∈| *gta-der* $\mathcal{A}$ *t* ⟹ (∃ *p qs*. *run* $\mathcal{A}$ (*GFun* (*p*, *q*) *qs*) *t*) **unfolding** *gta-der-def*
proof (*induct rule*: *ta-der-gterm-induct*)
  case (*GFun f ts ps p q*)
  from *GFun*(*5*) *Ex-list-of-length-P*[*of length ts* λ *qs i*. *run* $\mathcal{A}$ (*GFun* (*fst qs*, *ps* ! *i*) (*snd qs*)) (*ts* ! *i*)]
  obtain *qss* **where** *mid*: *length qss* = *length ts* ∀ *i* < *length ts* .*run* $\mathcal{A}$ (*GFun* (*fst* (*qss* ! *i*), *ps* ! *i*) (*snd* (*qss* ! *i*))) (*ts* ! *i*)
    by *auto*
  have [*simp*]: *map* (*ex-comp-state* ∘ (λ(*qs*, *y*). *GFun* (*fst y*, *qs*) (*snd y*))) (*zip ps qss*) = *ps* **using** *GFun*(*2*) *mid*(*1*)
    by (*intro nth-equalityI*) *auto*
  show *?case* **using** *mid GFun*(*1* − *4*)
    by (*intro exI*[*of* - *p*] *exI*[*of* - *map2* (λ *f args*. *GFun* (*fst args*, *f*) (*snd args*)) *ps qss*])
      (*auto intro*: *run.intros*)
qed

lemma *run-ta-der-ctxt-split1*:
  assumes *run* $\mathcal{A}$ *s t p* ∈ *gposs t*
  shows *ex-comp-state s* |∈| *ta-der* $\mathcal{A}$ (*ctxt-at-pos* (*term-of-gterm t*) *p*)⟨*Var* (*ex-comp-state* (*gsubt-at s p*))⟩
  using *assms*
proof (*induct p arbitrary*: *s t*)
  case (*Cons i p*)
  obtain *q f qs ts* **where** [*simp*]: *s* = *GFun q qs t* = *GFun f ts* **and** *l*: *length qs* = *length ts*
    using *run-argsD*[*OF Cons*(*2*)] **by** (*cases s*, *cases t*) *auto*
  from *Cons*(*2*, *3*) *l* **have** *ex-comp-state* (*qs* ! *i*) |∈| *ta-der* $\mathcal{A}$ (*ctxt-at-pos* (*term-of-gterm* (*ts* ! *i*)) *p*)⟨*Var* (*ex-comp-state* (*gsubt-at* (*qs* ! *i*) *p*))⟩
    by (*intro Cons*(*1*)) (*auto dest*: *run-argsD*)
  then show *?case* **using** *Cons*(*2*−) *l*
    by (*fastforce simp*: *nth-append-Cons min-def dest*: *run-root-rule run-argsD*
    intro!: *exI*[*of* - *map ex-comp-state* (*gargs s*)] *exI*[*of* - *ex-rule-state s*]
        run-to-comp-st-gta-der[*of* $\mathcal{A}$ *qs* ! *i ts* ! *i* **for** *i*, *unfolded comp-def gta-der-def*])
qed *auto*

lemma *run-ta-der-ctxt-split2*:
  assumes *run* $\mathcal{A}$ *s t p* ∈ *gposs t*
  shows *ex-comp-state s* |∈| *ta-der* $\mathcal{A}$ (*ctxt-at-pos* (*term-of-gterm t*) *p*)⟨*Var* (*ex-rule-state* (*gsubt-at s p*))⟩
proof (*cases ex-rule-state* (*gsubt-at s p*) = *ex-comp-state* (*gsubt-at s p*))
  case *False* **then show** *?thesis*
    using *run-root-rule*[*OF run-gsubt-cl*[*OF assms*]]
    by (*intro ta-der-eps-ctxt*[*OF run-ta-der-ctxt-split1*[*OF assms*]]) *auto*
qed (*auto simp*: *run-ta-der-ctxt-split1*[*OF assms*, *unfolded comp-def*])

end
theory *Tree-Automata-Det*
imports
Tree-Automata
begin

3.2 Powerset Construction for Tree Automata

The idea to treat states and transitions separately is from arXiv:1511.03595. Some parts of the implementation are also based on that paper. (The Algorithm corresponds roughly to the one in "Step 5")

Abstract Definitions and Correctness Proof

definition *ps-reachable-statesp* **where**
ps-reachable-statesp $\mathcal{A}$ *f ps* = (λ *q'*. $\exists$ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ *list-all2* (|∈|) *qs ps* ∧
(*q* = *q'* ∨ (*q*,*q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|))

lemma *ps-reachable-statespE*:
assumes *ps-reachable-statesp* $\mathcal{A}$ *f qs q*
obtains *ps p* **where** *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ *list-all2* (|∈|) *ps qs* (*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|)
using *assms* **unfolding** *ps-reachable-statesp-def*
by *auto*

lemma *ps-reachable-statesp-*$\mathcal{Q}$:
ps-reachable-statesp $\mathcal{A}$ *f ps q* ⟹ *q* |∈| $\mathcal{Q}$ $\mathcal{A}$
by (*auto simp*: *ps-reachable-statesp-def simp flip*: *dest*: *rule-statesD eps-trancl-statesD*)

lemma *finite-Collect-ps-statep* [*simp*]:
finite (*Collect* (*ps-reachable-statesp* $\mathcal{A}$ *f ps*)) (**is** *finite ?S*)
by (*intro finite-subset*[*of ?S fset* ($\mathcal{Q}$ $\mathcal{A}$)])
(*auto simp*: *ps-reachable-statesp-*$\mathcal{Q}$)
lemmas *finite-Collect-ps-statep-unfolded* [*simp*] = *finite-Collect-ps-statep*[*unfolded ps-reachable-statesp-def*, *simplified*]

definition *ps-reachable-states* $\mathcal{A}$ *f ps* ≡ *fCollect* (*ps-reachable-statesp* $\mathcal{A}$ *f ps*)

lemmas *ps-reachable-states-simp* = *ps-reachable-statesp-def ps-reachable-states-def*

lemma *ps-reachable-states-fmember*:
q' |∈| *ps-reachable-states* $\mathcal{A}$ *f ps* ⟷ ($\exists$ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧
list-all2 (|∈|) *qs ps* ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|))
by (*auto simp*: *ps-reachable-states-simp*)

lemma *ps-reachable-statesI*:
assumes *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ *list-all2* (|∈|) *ps qs* (*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|)

shows $p \mid\in\mid$ *ps-reachable-states* $\mathcal{A}$ *f qs*
using *assms* **unfolding** *ps-reachable-states-simp*
by *auto*

lemma *ps-reachable-states-sig*:
ps-reachable-states $\mathcal{A}$ *f ps* $\neq \{||\} \Longrightarrow$ (*f*, *length ps*) $\mid\in\mid$ *ta-sig* $\mathcal{A}$
by (*auto simp*: *ps-reachable-states-simp ta-sig-def image-iff intro*!: *bexI dest*!: *list-all2-lengthD*)

A set of "powerset states" is complete if it is sufficient to capture all (non)deterministic derivations.

definition *ps-states-complete-it* :: ('*a*, '*b*) *ta* ⇒ '*a FSet-Lex-Wrapper fset* ⇒ '*a FSet-Lex-Wrapper fset* ⇒ *bool*
where *ps-states-complete-it* $\mathcal{A}$ *Q Qnext* ≡
$\forall$*f ps*. *fset-of-list ps* $\mid\subseteq\mid$ *Q* ∧ *ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*) ≠ {||} ⟶ *Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*)) $\mid\in\mid$ *Qnext*

lemma *ps-states-complete-itD*:
ps-states-complete-it $\mathcal{A}$ *Q Qnext* $\Longrightarrow$ *fset-of-list ps* $\mid\subseteq\mid$ *Q* $\Longrightarrow$
ps-reachable-states $\mathcal{A}$ *f* (*map ex ps*) ≠ {||} $\Longrightarrow$ *Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*)) $\mid\in\mid$ *Qnext*
unfolding *ps-states-complete-it-def* **by** *blast*

abbreviation *ps-states-complete* $\mathcal{A}$ *Q* ≡ *ps-states-complete-it* $\mathcal{A}$ *Q Q*

The least complete set of states

inductive-set *ps-states-set* **for** $\mathcal{A}$ **where**
$\forall$ *p* ∈ *set ps*. *p* ∈ *ps-states-set* $\mathcal{A}$ $\Longrightarrow$ *ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*) ≠ {||} $\Longrightarrow$
Wrapp (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*)) ∈ *ps-states-set* $\mathcal{A}$

lemma *ps-states-Pow*:
ps-states-set $\mathcal{A}$ ⊆ *fset* (*Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$))
proof −
{fix *q* **assume** *q* ∈ *ps-states-set* $\mathcal{A}$ **then have** *q* ∈ *fset* (*Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$))
by *induct* (*auto simp*: *ps-reachable-statesp-*$\mathcal{Q}$ *ps-reachable-states-def image-iff*)**}**
then show *?thesis* **by** *blast*
qed

context
includes *fset.lifting*
begin
lift-definition *ps-states* :: ('*a*, '*b*) *ta* ⇒ '*a FSet-Lex-Wrapper fset* **is** *ps-states-set*
by (*auto intro*: *finite-subset*[*OF ps-states-Pow*])

lemma *ps-states*: *ps-states* $\mathcal{A}$ $\mid\subseteq\mid$ *Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$) **using** *ps-states-Pow*
by (*simp add*: *ps-states-Pow less-eq-fset.rep-eq ps-states.rep-eq*)

lemmas *ps-states-cases* = *ps-states-set.cases*[*Transfer.transferred*]
lemmas *ps-states-induct* = *ps-states-set.induct*[*Transfer.transferred*]
lemmas *ps-states-simps* = *ps-states-set.simps*[*Transfer.transferred*]
lemmas *ps-states-intros*= *ps-states-set.intros*[*Transfer.transferred*]
end

lemma *ps-states-complete*:
  ps-states-complete $\mathcal{A}$ (*ps-states* $\mathcal{A}$)
  unfolding *ps-states-complete-it-def*
  by (*auto intro*: *ps-states-intros*)

lemma *ps-states-least-complete*:
  assumes *ps-states-complete-it* $\mathcal{A}$ *Q Qnext Qnext* $|\subseteq|$ *Q*
    shows *ps-states* $\mathcal{A}$ $|\subseteq|$ *Q*
proof *standard*
  fix *q* **assume** *ass*: *q* $|\in|$ *ps-states* $\mathcal{A}$ **then show** *q* $|\in|$ *Q*
    using *ps-states-complete-itD*[*OF assms*(*1*)] *fsubsetD*[*OF assms*(*2*)]
    by (*induct rule*: *ps-states-induct*[*of* - $\mathcal{A}$]) (*fastforce intro*: *ass*)+
qed

definition *ps-rulesp* :: ('*a*, '*b*) *ta* ⇒ '*a FSet-Lex-Wrapper fset* ⇒ ('*a FSet-Lex-Wrapper*, '*b*) *ta-rule* ⇒ *bool* **where**
  ps-rulesp $\mathcal{A}$ *Q* = (λ *r*. ∃ *f ps p*. *r* = *TA-rule f ps* (*Wrapp p*) ∧ *fset-of-list ps* $|\subseteq|$ *Q* ∧
    p = *ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*) ∧ *p* ≠ {||})

definition *ps-rules* **where**
  ps-rules $\mathcal{A}$ *Q* ≡ *fCollect* (*ps-rulesp* $\mathcal{A}$ *Q*)

lemma *finite-ps-rulesp* [*simp*]:
  finite (*Collect* (*ps-rulesp* $\mathcal{A}$ *Q*)) (**is** *finite ?S*)
proof −
  let *?Q* = *fset* (*Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$) $|\cup|$ *Q*) **let** *?sig* = *fset* (*ta-sig* $\mathcal{A}$)
  define *args* **where** *args* ≡ ⋃ (*f*,*n*) ∈ *?sig*. {*qs*| *qs*. *set qs* ⊆ *?Q* ∧ *length qs* = *n*}
  define *bound* **where** *bound* ≡ ⋃(*f*,-) ∈ *?sig*. ⋃ *q* ∈ *?Q*. ⋃ *qs* ∈ *args*. {*TA-rule f qs q*}
  have *finite*: *finite ?Q finite ?sig* **by** (*auto intro*: *finite-subset*)
  then have *finite args* **using** *finite-lists-length-eq*[*OF* ‹*finite ?Q*›]
    by (*force simp*: *args-def*)
   with *finite* **have** *finite bound* **unfolding** *bound-def* **by** (*auto simp only*: *finite-UN*)
  moreover have *Collect* (*ps-rulesp* $\mathcal{A}$ *Q*) ⊆ *bound*
  proof *standard*
    fix *r* **assume** ∗: *r* ∈ *Collect* (*ps-rulesp* $\mathcal{A}$ *Q*)
    obtain *f ps p* **where** *r*[*simp*]: *r* = *TA-rule f ps p* **by** (*cases r*)
     from ∗ **obtain** *qs q* **where** *TA-rule f qs q* $|\in|$ *rules* $\mathcal{A}$ **and** *len*: *length ps* = *length qs*
      unfolding *ps-rulesp-def ps-reachable-states-simp*

using *list-all2-lengthD* **by** *fastforce*
from *this* **have** *sym*: $(f, length\ qs) \in ?sig$
by *auto*
moreover from $*$ **have** *set ps* $\subseteq$ *?Q* **unfolding** *ps-rulesp-def*
by (*auto simp flip*: *fset-of-list-elem simp*: *ps-reachable-statesp-def*)
ultimately have *ps*: $ps \in args$
by (*auto simp only*: *args-def UN-iff intro*!: *bexI*[*of* - (f, *length qs*)] *len*)
from $*$ **have** $p \in ?Q$ **unfolding** *ps-rulesp-def ps-reachable-states-def*
using *ps-reachable-statesp-*$\mathcal{Q}$
by (*fastforce simp add*: *image-iff*)
with *ps sym* **show** $r \in bound$
by (*auto simp only*: *r bound-def UN-iff intro*!: *bexI*[*of* - (f, *length qs*)] *bexI*[*of* - *p*] *bexI*[*of* - *ps*])
qed
ultimately show *?thesis* **by** (*blast intro*: *finite-subset*)
qed

lemmas *finite-ps-rulesp-unfolded* = *finite-ps-rulesp*[*unfolded ps-rulesp-def*, *simplified*]

lemmas *ps-rulesI* [*intro*!] = *fCollect-memberI*[*OF finite-ps-rulesp*]

lemma *ps-rules-states*:
rule-states (*fCollect* (*ps-rulesp* $\mathcal{A}$ *Q*)) |⊆| (*Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$) |∪| *Q*)
by (*auto simp*: *ps-rulesp-def rule-states-def ps-reachable-states-def ps-reachable-statesp-*$\mathcal{Q}$) *blast*

definition *ps-ta* :: (′*q*, ′*f*) *ta* ⇒ (′*q FSet-Lex-Wrapper*, ′*f*) *ta* **where**
ps-ta $\mathcal{A}$ = (*let Q* = *ps-states* $\mathcal{A}$ *in*
TA (*ps-rules* $\mathcal{A}$ *Q*) {||})

definition *ps-ta-*Q_f :: ′*q fset* ⇒ (′*q*, ′*f*) *ta* ⇒ ′*q FSet-Lex-Wrapper fset* **where**
ps-ta-Q_f *Q* $\mathcal{A}$ = (*let Q*′ = *ps-states* $\mathcal{A}$ *in*
ffilter (λ *S*. *Q* |∩| (*ex S*) ≠ {||}) *Q*′)

lemma *ps-rules-sound*:
assumes *p* |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*)
shows *ex p* |⊆| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **using** *assms*
proof (*induction rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
then have *IH*: ∀ *i* < *length ts*. *ex* (*ps* ! *i*) |⊆| *gta-der* $\mathcal{A}$ (*ts* ! *i*) **unfolding** *gta-der-def* **by** *auto*
show *?case*
proof *standard*
fix *r* **assume** *r* |∈| *ex q*
with *GFun*(*1* − *3*) **obtain** *qs q*′ **where** *TA-rule f qs q*′ |∈| *rules* $\mathcal{A}$
q′ = *r* ∨ (*q*′, *r*) |∈| (*eps* $\mathcal{A}$)|$^+$| *list-all2* (|∈|) *qs* (*map ex ps*)
by (*auto simp*: *Let-def ps-ta-def ps-rulesp-def ps-reachable-states-simp ps-rules-def*)
then show *r* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*GFun f ts*))

```
    using GFun(2) IH unfolding gta-der-def
   by (force dest!: fsubsetD intro!: exI[of - q'] exI[of - qs] simp: list-all2-conv-all-nth)
  qed
qed

lemma ps-ta-nt-empty-set:
  TA-rule f qs (Wrapp {||}) |∈| rules (ps-ta 𝒜) ⟹ False
  by (auto simp: ps-ta-def ps-rulesp-def ps-rules-def)

lemma ps-rules-not-empty-reach:
  assumes Wrapp {||} |∈| ta-der (ps-ta 𝒜) (term-of-gterm t)
  shows False using assms
proof (induction t)
  case (GFun f ts)
  then show ?case using ps-ta-nt-empty-set[of f - 𝒜]
    by (auto simp: ps-ta-def)
qed

lemma ps-rules-complete:
  assumes q |∈| ta-der 𝒜 (term-of-gterm t)
  shows ∃ p. q |∈| ex p ∧ p |∈| ta-der (ps-ta 𝒜) (term-of-gterm t) ∧ p |∈| ps-states
𝒜 using assms
proof (induction rule: ta-der-gterm-induct)
  let ?P = λt q p. q |∈| ex p ∧ p |∈| ta-der (ps-ta 𝒜) (term-of-gterm t) ∧ p |∈|
ps-states 𝒜
  case (GFun f ts ps p q)
  then have ∀ i. ∃ p. i < length ts ⟶ ?P (ts ! i) (ps ! i) p by auto
  with choice[OF this] obtain psf where ps: ∀ i < length ts. ?P (ts ! i) (ps ! i)
(psf i) by auto
  define qs where qs = map psf [0 ..< length ts]
  let ?p = ps-reachable-states 𝒜 f (map ex qs)
  from ps have in-Q: fset-of-list qs |⊆| ps-states 𝒜
    by (auto simp: qs-def fset-of-list-elem)
  from ps GFun(2) have all: list-all2 (|∈|) ps (map ex qs)
    by (auto simp: list-all2-conv-all-nth qs-def)
  then have in-p: q |∈| ?p using GFun(1, 3)
    unfolding ps-reachable-statesp-def ps-reachable-states-def by auto
 then have rule: TA-rule f qs (Wrapp ?p) |∈| ps-rules 𝒜 (ps-states 𝒜) using in-Q
unfolding ps-rules-def
    by (intro ps-rulesI) (auto simp: ps-rulesp-def)
  from in-Q in-p have Wrapp ?p |∈| (ps-states 𝒜)
   by (auto intro!: ps-states-complete[unfolded ps-states-complete-it-def, rule-format])
  with in-p ps rule show ?case
    by (auto intro!: exI[of - Wrapp ?p] exI[of - qs] simp: ps-ta-def qs-def)
qed

lemma ps-ta-eps[simp]: eps (ps-ta 𝒜) = {||} by (auto simp: Let-def ps-ta-def)

lemma ps-ta-det[iff]: ta-det (ps-ta 𝒜) by (auto simp: Let-def ps-ta-def ta-det-def
```

ps-rulesp-def ps-rules-def)

lemma *ps-gta-lang*:
gta-lang (*ps-ta-*Q_f Q $\mathcal{A}$) (*ps-ta* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$ (**is** *?R* = *?L*)
proof *standard*
show *?L* ⊆ *?R* **proof** *standard*
fix *t* **assume** *t* ∈ *?L*
then obtain *q* **where** *q-res*: *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **and** *q-final*: *q* |∈| *Q*
by *auto*
from *ps-rules-complete*[*OF q-res*] **obtain** *p* **where**
p |∈| *ps-states* $\mathcal{A}$ *q* |∈| *ex p p* |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*)
by *auto*
moreover with *q-final* **have** *p* |∈| *ps-ta-*Q_f Q $\mathcal{A}$
by (*auto simp*: *ps-ta-*Q_f*-def*)
ultimately show *t* ∈ *?R* **by** *auto*
qed
show *?R* ⊆ *?L* **proof** *standard*
fix *t* **assume** *t* ∈ *?R*
then obtain *p* **where**
p-res: *p* |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*) **and** *p-final*: *p* |∈| *ps-ta-*Q_f Q $\mathcal{A}$
by (*auto simp*: *ta-lang-def*)
from *ps-rules-sound*[*OF p-res*] **have** *ex p* |⊆| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
by *auto*
moreover from *p-final* **obtain** *q* **where** *q* |∈| *ex p q* |∈| *Q* **by** (*auto simp*: *ps-ta-*Q_f*-def*)
ultimately show *t* ∈ *?L* **by** *auto*
qed
qed

definition *ps-reg* **where**
ps-reg R = *Reg* (*ps-ta-*Q_f (*fin R*) (*ta R*)) (*ps-ta* (*ta R*))

lemma $\mathcal{L}$*-ps-reg*:
$\mathcal{L}$ (*ps-reg R*) = $\mathcal{L}$ *R*
by (*auto simp*: $\mathcal{L}$*-def ps-gta-lang ps-reg-def*)

lemma *ps-ta-states*: $\mathcal{Q}$ (*ps-ta* $\mathcal{A}$) |⊆| *Wrapp* |`| *fPow* ($\mathcal{Q}$ $\mathcal{A}$)
using *ps-rules-states ps-states* **unfolding** *ps-ta-def* $\mathcal{Q}$*-def*
by (*auto simp*: *Let-def ps-rules-def*) *force*

lemma *ps-ta-states′*: *ex* |`| $\mathcal{Q}$ (*ps-ta* $\mathcal{A}$) |⊆| *fPow* ($\mathcal{Q}$ $\mathcal{A}$)
using *ps-ta-states*[*of* $\mathcal{A}$]
by *fastforce*

end
theory *Tree-Automata-Complement*
imports *Tree-Automata-Det*

begin

3.3 Complement closure of regular languages

definition *partially-completely-defined-on* **where**
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ $\longleftrightarrow$
($\forall$ *t*. *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$ $\longleftrightarrow$ ($\exists$ *q*. *q* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*)))

definition *sig-ta* **where**
sig-ta $\mathcal{F}$ = *TA* ((λ (*f*, *n*). *TA-rule f* (*replicate n* ()) ()) $|`|$ $\mathcal{F}$) {||}

lemma *sig-ta-rules-fmember*:
TA-rule f qs q $|\in|$ *rules* (*sig-ta* $\mathcal{F}$) $\longleftrightarrow$ ($\exists$ *n*. (*f*, *n*) $|\in|$ $\mathcal{F}$ $\wedge$ *qs* = *replicate n* ()
$\wedge$ *q* = ())
by (*auto simp*: *sig-ta-def fimage-iff fBex-def*)

lemma *sig-ta-completely-defined*:
partially-completely-defined-on (*sig-ta* $\mathcal{F}$) $\mathcal{F}$
proof −
{fix *t* **assume** *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$
then have () $|\in|$ *ta-der* (*sig-ta* $\mathcal{F}$) (*term-of-gterm t*)
proof (*induct t*)
case (*GFun f ts*)
then show *?case*
by (*auto simp*: *sig-ta-rules-fmember SUP-le-iff*
intro!: *exI*[*of* - *replicate* (*length ts*) ()])
qed}
moreover
{fix *t q* **assume** *q* $|\in|$ *ta-der* (*sig-ta* $\mathcal{F}$) (*term-of-gterm t*)
then have *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
from *GFun*(*1* − *4*) *GFun*(*5*)[*THEN subsetD*] **show** *?case*
by (*auto simp*: *sig-ta-rules-fmember dest*!: *in-set-idx*)
qed}
ultimately show *?thesis*
unfolding *partially-completely-defined-on-def*
by *blast*
qed

lemma *ta-der-fsubset-sig-ta-completely*:
assumes *ta-subset* (*sig-ta* $\mathcal{F}$) $\mathcal{A}$ *ta-sig* $\mathcal{A}$ $|\subseteq|$ $\mathcal{F}$
shows *partially-completely-defined-on* $\mathcal{A}$ $\mathcal{F}$
proof −
have *ta-der* (*sig-ta* $\mathcal{F}$) *t* $|\subseteq|$ *ta-der* $\mathcal{A}$ *t* **for** *t*
using *assms* **by** (*simp add*: *ta-der-mono'*)
then show *?thesis* **using** *sig-ta-completely-defined assms*(*2*)
by (*auto simp*: *partially-completely-defined-on-def*)
(*metis ffunas-gterm.rep-eq fin-mono ta-der-gterm-sig*)

qed

lemma *completely-definied-ps-taI*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ $\Longrightarrow$ *partially-completely-defined-on* (*ps-ta* $\mathcal{A}$) $\mathcal{F}$
unfolding *partially-completely-defined-on-def*
using *ps-rules-not-empty-reach*[*of* $\mathcal{A}$]
using *fsubsetD*[*OF ps-rules-sound*[*of - A*]] *ps-rules-complete*[*of - A*]
by (*metis FSet-Lex-Wrapper.collapse fsubsetI fsubset-fempty*)

lemma *completely-definied-ta-union1I*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ $\Longrightarrow$ *ta-sig* $\mathcal{B}$ $|\subseteq|$ $\mathcal{F}$ $\Longrightarrow$ $\mathcal{Q}$ $\mathcal{A}$ $|\cap|$ $\mathcal{Q}$ $\mathcal{B}$ = $\{||\}$ $\Longrightarrow$
partially-completely-defined-on (*ta-union* $\mathcal{A}$ $\mathcal{B}$) $\mathcal{F}$
unfolding *partially-completely-defined-on-def*
using *ta-union-der-disj-states'*[*of* $\mathcal{A}$ $\mathcal{B}$]
by (*auto simp*: *ta-union-der-disj-states*)
(*metis ffunas-gterm.rep-eq fsubset-trans less-eq-fset.rep-eq ta-der-gterm-sig*)

lemma *completely-definied-fmaps-statesI*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ $\Longrightarrow$ *finj-on f* ($\mathcal{Q}$ $\mathcal{A}$) $\Longrightarrow$ *partially-completely-defined-on* (*fmap-states-ta f* $\mathcal{A}$) $\mathcal{F}$
unfolding *partially-completely-defined-on-def*
using *fsubsetD*[*OF ta-der-fmap-states-ta-mono2*, *of f* $\mathcal{A}$]
using *ta-der-to-fmap-states-der*[*of - A - f*]
by (*auto simp*: *fimage-iff fBex-def*) *fastforce+*

lemma *det-completely-defined-complement*:
assumes *partially-completely-defined-on* $\mathcal{A}$ $\mathcal{F}$ *ta-det* $\mathcal{A}$
shows *gta-lang* ($\mathcal{Q}$ $\mathcal{A}$ $|-|$ *Q*) $\mathcal{A}$ = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − *gta-lang Q* $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
proof −
{fix *t* **assume** *t* ∈ *?Ls*
then obtain *p* **where** *p*: *p* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ *p* $|\notin|$ *Q p* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
by *auto*
from *ta-detE*[*OF assms*(*2*) *p*(*3*)] **have** ∀ *q*. *q* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟶ *q* = *p*
by *blast*
moreover have *funas-gterm t* ⊆ *fset* $\mathcal{F}$
using *p*(*3*) *assms*(*1*) **unfolding** *partially-completely-defined-on-def*
by (*auto simp*: *less-eq-fset.rep-eq ffunas-gterm.rep-eq*)
ultimately have *t* ∈ *?Rs* **using** *p*(*2*)
by (*auto simp*: $\mathcal{T}_G$*-equivalent-def*)**}**
moreover
{fix *t* **assume** *t* ∈ *?Rs*
then have *f*: *funas-gterm t* ⊆ *fset* $\mathcal{F}$ ∀ *q*. *q* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟶ *q* $|\notin|$ *Q*
by (*auto simp*: $\mathcal{T}_G$*-equivalent-def*)
from *f*(*1*) **obtain** *p* **where** *p* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **using** *assms*(*1*)
by (*force simp*: *partially-completely-defined-on-def*)

```
    then have t ∈ ?Ls using f(2)
      by (auto simp: gterm-ta-der-states intro: gta-langI[of p])}
  ultimately show ?thesis by blast
qed

lemma ta-der-gterm-sig-fset:
  q |∈| ta-der A (term-of-gterm t) ⟹ funas-gterm t ⊆ fset (ta-sig A)
  using ta-der-gterm-sig
  by (metis ffunas-gterm.rep-eq less-eq-fset.rep-eq)

definition filter-ta-sig where
  filter-ta-sig F A = TA (ffilter (λ r. (r-root r, length (r-lhs-states r)) |∈| F) (rules
A)) (eps A)

definition filter-ta-reg where
  filter-ta-reg F R = Reg (fin R) (filter-ta-sig F (ta R))

lemma filter-ta-sig:
  ta-sig (filter-ta-sig F A) |⊆| F
  by (auto simp: ta-sig-def filter-ta-sig-def)

lemma filter-ta-sig-lang:
  gta-lang Q (filter-ta-sig F A) = gta-lang Q A ∩ T_G (fset F) (is ?Ls = ?Rs)
proof −
  let ?A = filter-ta-sig F A
  {fix t assume t ∈ ?Ls
    then obtain q where q: q |∈| Q q |∈| ta-der ?A (term-of-gterm t)
      by auto
    then have funas-gterm t ⊆ fset F
        using subset-trans[OF ta-der-gterm-sig-fset[OF q(2)] filter-ta-sig[unfolded
less-eq-fset.rep-eq]]
      by blast
    then have t ∈ ?Rs using q
      by (auto simp: T_G-equivalent-def filter-ta-sig-def
            intro!: gta-langI[of q] ta-der-el-mono[where ?q = q and B = A and
A = ?A])}
  moreover
  {fix t assume ass: t ∈ ?Rs
    then have funas: funas-gterm t ⊆ fset F
      by (auto simp: T_G-equivalent-def)
    from ass obtain p where p: p |∈| Q p |∈| ta-der A (term-of-gterm t)
      by auto
    from this(2) funas have p |∈| ta-der ?A (term-of-gterm t)
    proof (induct rule: ta-der-gterm-induct)
      case (GFun f ts ps p q)
      then show ?case
        by (auto simp: filter-ta-sig-def SUP-le-iff intro!: exI[of - ps] exI[of - p])
    qed
    then have t ∈ ?Ls using p(1) by auto}
```

ultimately show *?thesis* **by** *blast*
qed

lemma $\mathcal{L}$*-filter-ta-reg*:
$\mathcal{L}$ (*filter-ta-reg* $\mathcal{F}$ $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ ∩ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
using *filter-ta-sig-lang*
by (*auto simp*: $\mathcal{L}$*-def filter-ta-reg-def*)

definition *sig-ta-reg* **where**
sig-ta-reg $\mathcal{F}$ = *Reg* {||} (*sig-ta* $\mathcal{F}$)

lemma $\mathcal{L}$*-sig-ta-reg*:
$\mathcal{L}$ (*sig-ta-reg* $\mathcal{F}$) = {}
by (*auto simp*: $\mathcal{L}$*-def sig-ta-reg-def*)

definition *complement-reg* **where**
complement-reg R $\mathcal{F}$ = (*let* $\mathcal{A}$ = *ps-reg* (*reg-union* (*sig-ta-reg* $\mathcal{F}$) R) *in*
Reg ($\mathcal{Q}_r$ $\mathcal{A}$ |−| *fin* $\mathcal{A}$) (*ta* $\mathcal{A}$))

lemma $\mathcal{L}$*-complement-reg*:
assumes *ta-sig* (*ta* $\mathcal{A}$) |⊆| $\mathcal{F}$
shows $\mathcal{L}$ (*complement-reg* $\mathcal{A}$ $\mathcal{F}$) = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − $\mathcal{L}$ $\mathcal{A}$
proof −
have $\mathcal{L}$ (*complement-reg* $\mathcal{A}$ $\mathcal{F}$) = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − $\mathcal{L}$ (*ps-reg* (*reg-union* (*sig-ta-reg* $\mathcal{F}$) $\mathcal{A}$))
unfolding $\mathcal{L}$*-def complement-reg-def* **using** *assms*
by (*auto simp*: *complement-reg-def Let-def ps-reg-def reg-union-def sig-ta-reg-def*
sig-ta-completely-defined finj-Inl-Inr
intro!: *det-completely-defined-complement completely-definied-ps-taI*
completely-definied-ta-union1I completely-definied-fmaps-statesI)
then show *?thesis*
by (*auto simp*: $\mathcal{L}$*-ps-reg* $\mathcal{L}$*-union* $\mathcal{L}$*-sig-ta-reg*)
qed

lemma $\mathcal{L}$*-complement-filter-reg*:
$\mathcal{L}$ (*complement-reg* (*filter-ta-reg* $\mathcal{F}$ $\mathcal{A}$) $\mathcal{F}$) = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − $\mathcal{L}$ $\mathcal{A}$
proof −
have ∗: *ta-sig* (*ta* (*filter-ta-reg* $\mathcal{F}$ $\mathcal{A}$)) |⊆| $\mathcal{F}$
by (*auto simp*: *filter-ta-reg-def filter-ta-sig*)
show *?thesis* **unfolding** $\mathcal{L}$*-complement-reg*[*OF* ∗] $\mathcal{L}$*-filter-ta-reg*
by *blast*
qed

definition *difference-reg* **where**
difference-reg R L = (*let* F = *ta-sig* (*ta* R) *in*
reg-intersect R (*trim-reg* (*complement-reg* (*filter-ta-reg* F L) F)))

lemma $\mathcal{L}$*-difference-reg*:
$\mathcal{L}$ (*difference-reg* R L) = $\mathcal{L}$ R − $\mathcal{L}$ L (**is** *?Ls* = *?Rs*)

unfolding *difference-reg-def Let-def* $\mathcal{L}$*-trim* $\mathcal{L}$*-intersect* $\mathcal{L}$*-complement-filter-reg*
using *reg-funas* **by** *blast*

end
theory *Tree-Automata-Pumping*
imports *Tree-Automata*
begin

3.4 Pumping lemma

abbreviation *derivation-ctxt ts Cs* $\equiv$ *Suc* (*length Cs*) = *length ts* $\wedge$
($\forall$ *i* < *length Cs*. (*Cs* ! *i*) $\langle$*ts* ! *i*$\rangle$ = *ts* ! *Suc i*)

abbreviation *derivation-ctxt-st A ts Cs qs* $\equiv$ *length qs* = *length ts* $\wedge$ *Suc* (*length Cs*) = *length ts* $\wedge$
($\forall$ *i* < *length Cs*. *qs* ! *Suc i* $|\in|$ *ta-der A* (*Cs* ! *i*)$\langle$*Var* (*qs* ! *i*)$\rangle$)

abbreviation *derivation-sound A ts qs* $\equiv$ *length qs* = *length ts* $\wedge$
($\forall$ *i* < *length qs*. *qs* ! *i* $|\in|$ *ta-der A* (*ts* ! *i*))

definition *derivation A ts Cs qs* $\longleftrightarrow$ *derivation-ctxt ts Cs* $\wedge$
derivation-ctxt-st A ts Cs qs $\wedge$ *derivation-sound A ts qs*

lemma *ctxt-comp-lhs-not-hole*:
assumes *C* $\neq$ $\Box$
shows *C* $\circ_c$ *D* $\neq$ $\Box$
using *assms* **by** (*cases C*; *cases D*) *auto*

lemma *ctxt-comp-rhs-not-hole*:
assumes *D* $\neq$ $\Box$
shows *C* $\circ_c$ *D* $\neq$ $\Box$
using *assms* **by** (*cases C*; *cases D*) *auto*

lemma *fold-ctxt-comp-nt-empty-acc*:
assumes *D* $\neq$ $\Box$
shows *fold* ($\circ_c$) *Cs D* $\neq$ $\Box$
using *assms* **by** (*induct Cs arbitrary*: *D*) (*auto simp add*: *ctxt-comp-rhs-not-hole*)

lemma *fold-ctxt-comp-nt-empty*:
assumes *C* $\in$ *set Cs* **and** *C* $\neq$ $\Box$
shows *fold* ($\circ_c$) *Cs D* $\neq$ $\Box$ **using** *assms*
by (*induct Cs arbitrary*: *D*) (*auto simp*: *ctxt-comp-lhs-not-hole fold-ctxt-comp-nt-empty-acc*)

lemma *empty-ctxt-power* [*simp*]:
$\Box$ ^ *n* = $\Box$

by (*induct n*) *auto*

lemma *ctxt-comp-not-hole*:
assumes $C \neq \Box$ **and** $n \neq 0$
shows $C\hat{}n \neq \Box$
using *assms* **by** (*induct n arbitrary*: *C*) (*auto simp*: *ctxt-comp-lhs-not-hole*)

lemma *ctxt-comp-n-suc* [*simp*]:
shows $(C\hat{}(Suc\ n))\langle t\rangle = (C\hat{}n)\langle C\langle t\rangle\rangle$
by (*induct n arbitrary*: *C*) *auto*

lemma *ctxt-comp-reach*:
assumes p |∈| *ta-der A* $C\langle Var\ p\rangle$
shows p |∈| *ta-der A* $(C\hat{}n)\langle Var\ p\rangle$
using *assms* **by** (*induct n arbitrary*: *C*) (*auto intro*: *ta-der-ctxt*)

lemma *args-depth-less* [*simp*]:
assumes $u \in set\ ss$
shows *depth* $u <$ *depth* (*Fun f ss*) **using** *assms*
by (*cases ss*) (*auto simp*: *less-Suc-eq-le*)

lemma *subterm-depth-less*:
assumes $s \rhd t$
shows *depth* $t <$ *depth* s
using *assms* **by** (*induct s t rule*: *supt.induct*) (*auto intro*: *less-trans*)

lemma *poss-length-depth*:
shows $\exists\ p \in poss\ t.\ length\ p = depth\ t$
proof (*induct t*)
case (*Fun f ts*)
then show *?case*
proof (*cases ts*)
case [*simp*]: (*Cons a list*)
have $ts \neq [] \Longrightarrow \exists\ s.\ f\ s = Max\ (f\ `\ set\ ts) \wedge s \in set\ ts$ **for** *ts f*
using *Max-in*[*of f ` set ts*] **by** (*auto simp*: *image-iff*)
from *this*[*of ts depth*] **obtain** s **where** s: *depth* $s = Max\ (depth\ `\ set\ ts) \wedge s \in$ *set ts*
by *auto*
then show *?thesis* **using** *Fun*[*of s*] *in-set-idx*[*OF conjunct2*[*OF s*]]
by *fastforce*
qed *auto*
qed *auto*

lemma *poss-length-bounded-by-depth*:
assumes $p \in poss\ t$
shows *length* $p \leq$ *depth* t **using** *assms*

by (*induct t arbitrary*: *p*) (*auto intro*!: *Suc-leI*, *meson args-depth-less dual-order.strict-trans2 nth-mem*)

lemma *depth-ctxt-nt-hole-inc*:
assumes $C \neq \Box$
shows *depth t* < *depth* $C\langle t \rangle$ **using** *assms*
using *subterm-depth-less*[*of t* $C\langle t \rangle$]
by (*simp add*: *nectxt-imp-supt-ctxt subterm-depth-less*)

lemma *depth-ctxt-less-eq*:
depth t ≤ *depth* $C\langle t \rangle$ **using** *depth-ctxt-nt-hole-inc less-imp-le*
by (*cases C*, *simp*) *blast*

lemma *ctxt-comp-n-not-hole-depth-inc*:
assumes $C \neq \Box$
shows *depth* $(C^\frown n)\langle t \rangle$ < *depth* $(C^\frown(Suc\ n))\langle t \rangle$
using *assms* **by** (*induct n arbitrary*: *C t*) (*auto simp*: *ctxt-comp-not-hole depth-ctxt-nt-hole-inc*)

lemma *ctxt-comp-n-lower-bound*:
assumes $C \neq \Box$
shows *n* < *depth* $(C^\frown(Suc\ n))\langle t \rangle$
using *assms*
proof (*induct n arbitrary*: *C*)
case *0* **then show** *?case* **using** *ctxt-comp-not-hole depth-ctxt-nt-hole-inc gr-implies-not-zero*
by *blast*
next
case (*Suc n*) **then show** *?case* **using** *ctxt-comp-n-not-hole-depth-inc less-trans-Suc*
by *blast*
qed

lemma *ta-der-ctxt-n-loop*:
assumes *q* |∈| *ta-der* $\mathcal{A}$ *t q* |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ q \rangle$
shows *q* |∈| *ta-der* $\mathcal{A}$ $(C^\frown n)\langle t \rangle$
using *assms* **by** (*induct n*) (*auto simp*: *ta-der-ctxt*)

lemma *ctxt-compose-funs-ctxt* [*simp*]:
funs-ctxt $(C \circ_c D)$ = *funs-ctxt C* ∪ *funs-ctxt D*
by (*induct C arbitrary*: *D*) *auto*

lemma *ctxt-compose-vars-ctxt* [*simp*]:
vars-ctxt $(C \circ_c D)$ = *vars-ctxt C* ∪ *vars-ctxt D*
by (*induct C arbitrary*: *D*) *auto*

lemma *ctxt-power-funs-vars-0* [*simp*]:
assumes *n* = *0*
shows *funs-ctxt* $(C^\frown n)$ = {} *vars-ctxt* $(C^\frown n)$ = {}
using *assms* **by** *auto*

lemma *ctxt-power-funs-vars-n* [*simp*]:
assumes $n \neq 0$
shows *funs-ctxt* $(C\hat{}n)$ = *funs-ctxt* C *vars-ctxt* $(C\hat{}n)$ = *vars-ctxt* C
using *assms* **by** (*induct n arbitrary*: C, *auto*) *fastforce+*

fun *terms-pos* **where**
terms-pos s [] = [s]
| *terms-pos* s (p # ps) = *terms-pos* (s |- [p]) ps @ [s]

lemma *subt-at-poss* [*simp*]:
assumes a # $p \in$ *poss* s
shows $p \in$ *poss* (s |- [a])
using *assms* **by** (*metis append-Cons append-self-conv2 poss-append-poss*)

lemma *terms-pos-length* [*simp*]:
shows *length* (*terms-pos* t p) = *Suc* (*length* p)
by (*induct* p *arbitrary*: t) *auto*

lemma *terms-pos-last* [*simp*]:
assumes i = *length* p
shows *terms-pos* t p ! i = t **using** *assms*
by (*induct* p *arbitrary*: t) (*auto simp add*: *append-Cons-nth-middle*)

lemma *terms-pos-subterm*:
assumes $p \in$ *poss* t **and** $s \in$ *set* (*terms-pos* t p)
shows $t \trianglerighteq s$ **using** *assms*
using *assms*
proof (*induct* p *arbitrary*: t s)
case (*Cons* a p)
from *Cons*(*2*) **have** *st*: $t \trianglerighteq t$ |- [a] **by** *auto*
from *Cons*(*1*)[*of* t |- [a]] *Cons*(*2*−) **show** *?case*
using *supteq-trans*[*OF st*] **by** *fastforce*
qed *auto*

lemma *terms-pos-differ-subterm*:
assumes $p \in$ *poss* t **and** $i <$ *length* (*terms-pos* t p)
and $j <$ *length* (*terms-pos* t p) **and** $i < j$
shows *terms-pos* t p ! $i \lhd$ *terms-pos* t p ! j
using *assms*
proof (*induct* p *arbitrary*: t i j)
case (*Cons* a p)
from *Cons*(*2*−) **have** t |- [a] $\trianglerighteq$ *terms-pos* (t |- [a]) p ! i
by (*intro terms-pos-subterm*[*of* p]) *auto*
from *subterm.order.strict-trans1*[*OF this*, *of* t] *Cons*(*1*)[*of* t |- [a] i j] *Cons*(*2*−)
show *?case*

by (*cases j = length (a # p)*) (*force simp add*: *append-Cons-nth-middle append-Cons-nth-left*)+
qed *auto*

lemma *distinct-terms-pos*:
assumes *p ∈ poss t*
shows *distinct* (*terms-pos t p*) **using** *assms*
proof (*induct p arbitrary*: *t*)
case (*Cons a p*)
have ⋀*i*. *i < Suc* (*length p*) ⟹ *t ▷* (*terms-pos* (*t* |- [*a*]) *p*) ! *i*
using *terms-pos-differ-subterm*[*OF Cons*(*2*), *of* - *Suc* (*length p*)] **by** (*auto simp*: *nth-append*)
then show *?case* **using** *Cons*(*1*)[*of t* |- [*a*]] *Cons*(*2*−)
by (*auto simp*: *in-set-conv-nth*) (*metis supt-not-sym*)
qed *auto*

lemma *term-chain-depth*:
assumes *depth t = n*
shows ∃ *p ∈ poss t*. *length* (*terms-pos t p*) = (*n + 1*)
proof −
obtain *p* **where** *p*: *p ∈ poss t length p = depth t*
using *poss-length-depth*[*of t*] **by** *blast*
from *terms-pos-length*[*of t p*] *this* **show** *?thesis* **using** *assms*
by *auto*
qed

lemma *ta-der-derivation-chain-terms-pos-exist*:
assumes *p ∈ poss t* **and** *q* |∈| *ta-der A t*
shows ∃ *Cs qs*. *derivation A* (*terms-pos t p*) *Cs qs* ∧ *last qs = q*
using *assms*
proof (*induct p arbitrary*: *t q*)
case *Nil*
then show *?case* **by** (*auto simp*: *derivation-def intro*!: *exI*[*of* - [*q*]])
next
case (*Cons a p*)
from *Cons*(*2*) **have** *poss*: *p ∈ poss* (*t* |- [*a*]) **by** *auto*
from *Cons*(*2*) **obtain** *C* **where** *C*: *C*⟨*t* |- [*a*]⟩ = *t*
using *ctxt-at-pos-subt-at-id poss-Cons* **by** *blast*
from *C ta-der-ctxt-decompose Cons*(*3*) **obtain** *q′* **where**
res: *q′* |∈| *ta-der A* (*t* |- [*a*]) *q* |∈| *ta-der A C*⟨*Var q′*⟩
by *metis*
from *Cons*(*1*)[*OF* - *res*(*1*)] *Cons*(*2*−) *C* **obtain** *Cs qs* **where**
der: *derivation A* (*terms-pos* (*t* |- [*a*]) *p*) *Cs qs* ∧ *last qs = q′*
by (*auto simp del*: *terms-pos.simps*)
{fix *i* **assume** *i < Suc* (*length Cs*)
then have *derivation-ctxt* (*terms-pos* (*t* |- [*a*]) *p* @ [*t*]) (*Cs* @ [*C*])
using *der C*[*symmetric*] **unfolding** *derivation-def*
by (*cases i = length Cs*) (*auto simp*: *nth-append-Cons*)**}**

note *der-ctxt* = *this*
{fix *i* **assume** *i* < *Suc* (*length Cs*)
then have *derivation-ctxt-st A* (*terms-pos* (*t* |- [*a*]) *p* @ [*t*]) (*Cs* @ [*C*]) (*qs* @ [*q*])
using *der poss C res*(*2*) *last-conv-nth*[*of qs*]
by (*cases i* = *length Cs*, *auto 0 0 simp*: *derivation-def nth-append not-less less-Suc-eq*) *fastforce*+**}**
then show *?case* **using** *C poss res*(*1*) *der-ctxt der*
by (*auto simp*: *derivation-def intro*!: *exI*[*of* - *Cs* @ [*C*]] *exI*[*of* - *qs* @ [*q*]])
(*simp add*: *Cons.prems*(*2*) *nth-append-Cons*)
qed

lemma *derivation-ctxt-terms-pos-nt-empty*:
assumes *p* ∈ *poss t* **and** *derivation-ctxt* (*terms-pos t p*) *Cs* **and** *C* ∈ *set Cs*
shows *C* ≠ □
using *assms* **by** (*auto simp*: *in-set-conv-nth*)
(*metis Suc-mono assms*(*2*) *intp-actxt.simps*(*1*) *distinct-terms-pos lessI less-SucI less-irrefl-nat nth-eq-iff-index-eq*)

lemma *derivation-ctxt-terms-pos-sub-list-nt-empty*:
assumes *p* ∈ *poss t* **and** *derivation-ctxt* (*terms-pos t p*) *Cs*
and *i* < *length Cs* **and** *j* ≤ *length Cs* **and** *i* < *j*
shows *fold* (∘$_c$) (*take* (*j* − *i*) (*drop i Cs*)) □ ≠ □
proof −
have ∃ *C*. *C* ∈ *set* (*take* (*j* − *i*) (*drop i Cs*))
using *assms*(*3*−) *not-le* **by** *fastforce*
then obtain *C* **where** *w*: *C* ∈ *set* (*take* (*j* − *i*) (*drop i Cs*)) **by** *blast*
then have *C* ≠ □
by *auto* (*meson assms*(*1*, *2*) *derivation-ctxt-terms-pos-nt-empty in-set-dropD in-set-takeD*)
then show *?thesis* **by** (*auto simp*: *fold-ctxt-comp-nt-empty*[*OF w*])
qed

lemma *derivation-ctxt-comp-term*:
assumes *derivation-ctxt ts Cs*
and *i* < *length Cs* **and** *j* ≤ *length Cs* **and** *i* < *j*
shows (*fold* (∘$_c$) (*take* (*j* − *i*) (*drop i Cs*)) □)⟨*ts* ! *i*⟩ = *ts* ! *j*
using *assms*
proof (*induct j* − *i arbitrary*: *j i*)
case (*Suc x*)
then obtain *n* **where** *j* [*simp*]: *j* = *Suc n* **by** (*meson lessE*)
then have *r*: *x* = *n* − *i Suc n* − *i* = *1* + (*n* − *i*) **using** *Suc*(*2*, *6*) **by** *linarith*+
then show *?case* **using** *Suc*(*1*)[*OF r*(*1*)] *Suc*(*2*−) **unfolding** *j r*(*2*) *take-add*[*of n* − *i 1*]
by (*cases i* = *n*) (*auto simp*: *take-Suc-conv-app-nth*)
qed *auto*

lemma *derivation-ctxt-comp-states*:
assumes *derivation-ctxt-st A ts Cs qs*

and $i < length\ Cs$ **and** $j \leq length\ Cs$ **and** $i < j$
shows $qs\ !\ j\ |\in|\ ta\text{-}der\ A\ (fold\ (\circ_c)\ (take\ (j - i)\ (drop\ i\ Cs))\ \Box)\langle Var\ (qs\ !\ i)\rangle$
using *assms*
proof (*induct* $j - i$ *arbitrary*: $j\ i$)
case (*Suc* x)
then obtain n **where** j [*simp*]: $j = Suc\ n$ **by** (*meson lessE*)
then have r: $x = n - i\ Suc\ n - i = 1 + (n - i)$ **using** *Suc*(2, 6) **by** *linarith*+

then show *?case* **using** *Suc*(1)[*OF r*(1)] *Suc*(2−) **unfolding** $j\ r(2)$ *take-add*[*of* $n - i\ 1$]
by (*cases* $i = n$) (*auto simp*: *take-Suc-conv-app-nth ta-der-ctxt*)
qed *auto*

lemma *terms-pos-ground*:
assumes *ground t* **and** $p \in poss\ t$
shows $\forall\ s \in set\ (terms\text{-}pos\ t\ p).\ ground\ s$
using *terms-pos-subterm*[*OF assms*(2)] *subterm-eq-pres-ground*[*OF assms*(1)] **by** *simp*

lemma *list-card-smaller-contains-eq-elemens*:
assumes $length\ qs = n$ **and** $card\ (set\ qs) < n$
shows $\exists\ i < length\ qs.\ \exists\ j < length\ qs.\ i < j \wedge qs\ !\ i = qs\ !\ j$
using *assms* **by** *auto* (*metis distinct-card distinct-conv-nth linorder-neqE-nat*)

lemma *length-remdups-less-eq*:
assumes $set\ xs \subseteq set\ ys$
shows $length\ (remdups\ xs) \leq length\ (remdups\ ys)$ **using** *assms*
by (*auto simp*: *length-remdups-card-conv card-mono*)

lemma *pigeonhole-tree-automata*:
assumes $fcard\ (\mathcal{Q}\ A) < depth\ t$ **and** $q\ |\in|\ ta\text{-}der\ A\ t$ **and** *ground t*
shows $\exists\ C\ C2\ v\ p.\ C2 \neq \Box \wedge C\langle C2\langle v\rangle\rangle = t \wedge p\ |\in|\ ta\text{-}der\ A\ v \wedge$
$p\ |\in|\ ta\text{-}der\ A\ C2\langle Var\ p\rangle \wedge q\ |\in|\ ta\text{-}der\ A\ C\langle Var\ p\rangle$
proof −
obtain $p\ n$ **where** p: $p \in poss\ t\ \ depth\ t = n$ **and**
card: $fcard\ (\mathcal{Q}\ A) < n\ length\ (terms\text{-}pos\ t\ p) = (n + 1)$
using *assms*(1) *term-chain-depth* **by** *blast*
from *ta-der-derivation-chain-terms-pos-exist*[*OF p*(1) *assms*(2)] **obtain** *Cs qs* **where**
derivation: $derivation\ A\ (terms\text{-}pos\ t\ p)\ Cs\ qs \wedge last\ qs = q$ **by** *blast*
then have *d-ctxt*: *derivation-ctxt-st A* (*terms-pos t p*) *Cs qs derivation-ctxt* (*terms-pos t p*) *Cs*
by (*auto simp*: *derivation-def*)
then have *l*: $length\ Cs = length\ qs - 1$ **by** (*auto simp*: *derivation-def*)
from *derivation* **have** *sub*: $fset\text{-}of\text{-}list\ qs\ |\subseteq|\ \mathcal{Q}\ A\ length\ qs = length\ (terms\text{-}pos\ t\ p)$

unfolding *derivation-def*
using *ta-der-states*[*of A t* |- *i* **for** *i*] *terms-pos-ground*[*OF assms*(3) *p*(1)]
by *auto* (*metis derivation derivation-def gterm-of-term-inv gterm-ta-der-states in-fset-conv-nth nth-mem*)
then have ∃ *i* < *length* (*butlast qs*). ∃ *j* < *length* (*butlast qs*). *i* < *j* ∧ (*butlast qs*) ! *i* = (*butlast qs*) ! *j*
using *card*(1, 2) *assms*(1) *fcard-mono*[*OF sub*(1)] *length-remdups-less-eq*[*of butlast qs qs*]
by (*intro list-card-smaller-contains-eq-elemens*[*of butlast qs n*])
(*auto simp*: *card-set fcard-fset in-set-butlastD subsetI*
intro!: *le-less-trans*[*of length* (*remdups* (*butlast qs*)) *fcard* (*Q A*) *length p*])
then obtain *i j* **where** *len*: *i* < *length Cs j* < *length Cs* **and** *less*: *i* < *j* **and** *st*: *qs* ! *i* = *qs* ! *j*
unfolding *l length-butlast* **by** (*auto simp*: *nth-butlast*)
then have *gt-0*: *0* < *length Cs* **and** *gt-j*: *0* < *j* **using** *len less less-trans* **by** *auto*
have *fold* (∘$_c$) (*take* (*j* − *i*) (*drop i Cs*)) □ ≠ □
using *derivation-ctxt-terms-pos-sub-list-nt-empty*[*OF p*(1) *d-ctxt*(2) *len*(1) *order.strict-implies-order*[*OF len*(2)] *less*] .
moreover have (*fold* (∘$_c$) (*take* (*length Cs* − *j*) (*drop j Cs*)) □)⟨*terms-pos t p* ! *j*⟩ = *terms-pos t p* ! *length Cs*
using *derivation-ctxt-comp-term*[*OF d-ctxt*(2) *len*(2) - *len*(2)] *len*(2) **by** *auto*
moreover have (*fold* (∘$_c$) (*take* (*j* − *i*) (*drop i Cs*)) □)⟨*terms-pos t p* ! *i*⟩ = *terms-pos t p* ! *j*
using *derivation-ctxt-comp-term*[*OF d-ctxt*(2) *len*(1) - *less*] *len*(2) **by** *auto*
moreover have *qs* ! *j* |∈| *ta-der A* (*terms-pos t p* ! *i*) **using** *derivation len*
by (*auto simp*: *derivation-def st*[*symmetric*])
moreover have *qs* ! *j* |∈| *ta-der A* (*fold* (∘$_c$) (*take* (*j* − *i*) (*drop i Cs*)) □)⟨*Var* (*qs* ! *i*)⟩
using *derivation-ctxt-comp-states*[*OF d-ctxt*(1) *len*(1) - *less*] *len*(2) *st* **by** *simp*
moreover have *q* |∈| *ta-der A* (*fold* (∘$_c$) (*take* (*length Cs* − *j*) (*drop j Cs*)) □)⟨*Var* (*qs* ! *j*)⟩
using *derivation-ctxt-comp-states*[*OF d-ctxt*(1) *len*(2) - *len*(2)] *conjunct2*[*OF derivation*]
by (*auto simp*: *l sub*(2)) (*metis Suc-inject Zero-not-Suc d-ctxt*(1) *l last-conv-nth list.size*(3) *terms-pos-length*)
ultimately show *?thesis* **using** *st d-ctxt*(1) **by** (*metis Suc-inject terms-pos-last terms-pos-length*)
qed

end
theory *Myhill-Nerode*
imports *Tree-Automata Ground-Ctxt*
begin

3.5 Myhill Nerode characterization for regular tree languages

lemma *ground-ctxt-apply-pres-der*:
assumes *ta-der* $\mathcal{A}$ (*term-of-gterm s*) = *ta-der* $\mathcal{A}$ (*term-of-gterm t*)

shows *ta-der* $\mathcal{A}$ (*term-of-gterm* $C\langle s\rangle_G$) = *ta-der* $\mathcal{A}$ (*term-of-gterm* $C\langle t\rangle_G$) **using** *assms*
by (*induct C*) (*auto*, (*metis append-Cons-nth-not-middle nth-append-length*)+)

locale *myhill-nerode* =
fixes $\mathcal{F}$ $\mathcal{L}$ **assumes** *term-subset*: $\mathcal{L} \subseteq \mathcal{T}_G\ \mathcal{F}$
begin

definition *myhill* (‹- $\equiv_{\mathcal{L}}$ -›) **where**
myhill s t $\equiv s \in \mathcal{T}_G\ \mathcal{F} \wedge t \in \mathcal{T}_G\ \mathcal{F} \wedge (\forall\ C.\ C\langle s\rangle_G \in \mathcal{L} \wedge C\langle t\rangle_G \in \mathcal{L} \vee C\langle s\rangle_G \notin \mathcal{L} \wedge C\langle t\rangle_G \notin \mathcal{L})$

lemma *myhill-sound*: $s \equiv_{\mathcal{L}} t \Longrightarrow s \in \mathcal{T}_G\ \mathcal{F}\ \ s \equiv_{\mathcal{L}} t \Longrightarrow t \in \mathcal{T}_G\ \mathcal{F}$
unfolding *myhill-def* **by** *auto*

lemma *myhill-refl* [*simp*]: $s \in \mathcal{T}_G\ \mathcal{F} \Longrightarrow s \equiv_{\mathcal{L}} s$
unfolding *myhill-def* **by** *auto*

lemma *myhill-symmetric*: $s \equiv_{\mathcal{L}} t \Longrightarrow t \equiv_{\mathcal{L}} s$
unfolding *myhill-def* **by** *auto*

lemma *myhill-trans* [*trans*]:
$s \equiv_{\mathcal{L}} t \Longrightarrow t \equiv_{\mathcal{L}} u \Longrightarrow s \equiv_{\mathcal{L}} u$
unfolding *myhill-def* **by** *auto*

abbreviation *myhill-r* (‹$MN_{\mathcal{L}}$›) **where**
myhill-r $\equiv \{(s, t) \mid s\ t.\ s \equiv_{\mathcal{L}} t\}$

lemma *myhill-equiv*:
equiv ($\mathcal{T}_G\ \mathcal{F}$) $MN_{\mathcal{L}}$
apply (*intro equivI*) **apply** (*auto simp*: *myhill-sound myhill-symmetric sym-def trans-def refl-on-def*)
using *myhill-trans* **by** *blast*

lemma *rtl-der-image-on-myhill-inj*:
assumes *gta-lang* $Q_f\ \mathcal{A} = \mathcal{L}$
shows *inj-on* ($\lambda\ X.$ *gta-der* $\mathcal{A}$ ‘ X) ($\mathcal{T}_G\ \mathcal{F}$ // $MN_{\mathcal{L}}$) (**is** *inj-on ?D ?R*)
proof −
{fix $S\ T$ **assume** *eq-rel*: $S \in ?R\ T \in ?R\ ?D\ S = ?D\ T$
have $\bigwedge s\ t.\ s \in S \Longrightarrow t \in T \Longrightarrow s \equiv_{\mathcal{L}} t$
proof −
fix $s\ t$ **assume** *mem*: $s \in S\ t \in T$
then obtain t' **where** *res*: $t' \in T$ *gta-der* $\mathcal{A}\ s =$ *gta-der* $\mathcal{A}\ t'$ **using** *eq-rel*(*3*)
by (*metis image-iff*)
from *res*(*1*) *mem* **have** $s \in \mathcal{T}_G\ \mathcal{F}\ t \in \mathcal{T}_G\ \mathcal{F}\ t' \in \mathcal{T}_G\ \mathcal{F}$ **using** *eq-rel*(*1*, *2*)
using *in-quotient-imp-subset myhill-equiv* **by** *blast*+
then have $s \equiv_{\mathcal{L}} t'$ **using** *assms res ground-ctxt-apply-pres-der*[*of* $\mathcal{A}\ s$]
by (*auto simp*: *myhill-def gta-der-def simp flip*: *ctxt-of-gctxt-apply*
elim!: *gta-langE intro*: *gta-langI*)

moreover have $t' \equiv_{\mathcal{L}} t$ **using** *quotient-eq-iff*[*OF myhill-equiv eq-rel*(2) *eq-rel*(2) *res*(1) *mem*(2)]
by *simp*
ultimately show $s \equiv_{\mathcal{L}} t$ **using** *myhill-trans* **by** *blast*
qed
then have $\bigwedge s\ t.\ s \in S \Longrightarrow t \in T \Longrightarrow (s, t) \in MN_{\mathcal{L}}$ **by** *blast*
then have $S = T$ **using** *quotient-eq-iff*[*OF myhill-equiv eq-rel*(1, 2)]
using *eq-rel*(3) **by** *fastforce*}
then show *inj*: *inj-on ?D ?R* **by** (*meson inj-onI*)
qed

lemma *rtl-implies-finite-indexed-myhill-relation*:
assumes *gta-lang* $Q_f\ \mathcal{A} = \mathcal{L}$
shows *finite* $(\mathcal{T}_G\ \mathcal{F}\ //\ MN_{\mathcal{L}})$ (**is** *finite ?R*)
proof −
let *?D* = $\lambda\ X.$ *gta-der* $\mathcal{A}$ ‘ *X*
have *image*: *?D* ‘ *?R* ⊆ *Pow* (*fset* (*fPow* ($\mathcal{Q}\ \mathcal{A}$)))) **unfolding** *gta-der-def*
by (*meson PowI fPowI ground-ta-der-states ground-term-of-gterm image-subsetI*)
then have *finite* (*Pow* (*fset* (*fPow* ($\mathcal{Q}\ \mathcal{A}$)))) **by** *simp*
then have *finite* (*?D* ‘ *?R*) **using** *finite-subset*[*OF image*] **by** *fastforce*
then show *?thesis* **using** *finite-image-iff*[*OF rtl-der-image-on-myhill-inj*[*OF assms*]]
by *blast*
qed

end

end
theory *GTT*
imports *Tree-Automata Ground-Closure*
begin

4 Ground Tree Transducers (GTT)

type-synonym $('q, 'f)$ *gtt* = $('q, 'f)$ *ta* × $('q, 'f)$ *ta*

abbreviation *gtt-rules* **where**
gtt-rules $\mathcal{G}$ ≡ *rules* (*fst* $\mathcal{G}$) |∪| *rules* (*snd* $\mathcal{G}$)
abbreviation *gtt-eps* **where**
gtt-eps $\mathcal{G}$ ≡ *eps* (*fst* $\mathcal{G}$) |∪| *eps* (*snd* $\mathcal{G}$)
definition *gtt-states* **where**
gtt-states $\mathcal{G}$ = $\mathcal{Q}$ (*fst* $\mathcal{G}$) |∪| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
abbreviation *gtt-syms* **where**
gtt-syms $\mathcal{G}$ ≡ *ta-sig* (*fst* $\mathcal{G}$) |∪| *ta-sig* (*snd* $\mathcal{G}$)
definition *gtt-interface* **where**
gtt-interface $\mathcal{G}$ = $\mathcal{Q}$ (*fst* $\mathcal{G}$) |∩| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
definition *gtt-eps-free* **where**
gtt-eps-free $\mathcal{G}$ = (*eps-free* (*fst* $\mathcal{G}$), *eps-free* (*snd* $\mathcal{G}$))

definition *is-gtt-eps-free* :: $('q, 'f)$ *ta* × $('p, 'g)$ *ta* ⇒ *bool* **where**

is-gtt-eps-free $\mathcal{G} \longleftrightarrow$ *eps* (*fst* $\mathcal{G}$) = {||} $\wedge$ *eps* (*snd* $\mathcal{G}$) = {||}

anchored language accepted by a GTT

definition *agtt-lang* :: ('*q*, '*f*) *gtt* $\Rightarrow$ '*f gterm rel* **where**
agtt-lang $\mathcal{G}$ = {(*t*, *u*) |*t u q*. *q* |∈| *gta-der* (*fst* $\mathcal{G}$) *t* $\wedge$ *q* |∈| *gta-der* (*snd* $\mathcal{G}$) *u*}

lemma *agtt-langI*:
q |∈| *gta-der* (*fst* $\mathcal{G}$) *s* $\Longrightarrow$ *q* |∈| *gta-der* (*snd* $\mathcal{G}$) *t* $\Longrightarrow$ (*s*, *t*) $\in$ *agtt-lang* $\mathcal{G}$
by (*auto simp*: *agtt-lang-def*)

lemma *agtt-langE*:
assumes (*s*, *t*) $\in$ *agtt-lang* $\mathcal{G}$
obtains *q* **where** *q* |∈| *gta-der* (*fst* $\mathcal{G}$) *s q* |∈| *gta-der* (*snd* $\mathcal{G}$) *t*
using *assms* **by** (*auto simp*: *agtt-lang-def*)

lemma *converse-agtt-lang*:
(*agtt-lang* $\mathcal{G}$)$^{-1}$ = *agtt-lang* (*prod.swap* $\mathcal{G}$)
by (*auto simp*: *agtt-lang-def*)

lemma *agtt-lang-swap*:
agtt-lang (*prod.swap* $\mathcal{G}$) = *prod.swap* ‘ *agtt-lang* $\mathcal{G}$
by (*auto simp*: *agtt-lang-def*)

language accepted by a GTT

abbreviation *gtt-lang* :: ('*q*, '*f*) *gtt* $\Rightarrow$ '*f gterm rel* **where**
gtt-lang $\mathcal{G}$ $\equiv$ *gmctxt-cl UNIV* (*agtt-lang* $\mathcal{G}$)

lemma *gtt-lang-join*:
q |∈| *gta-der* (*fst* $\mathcal{G}$) *s* $\Longrightarrow$ *q* |∈| *gta-der* (*snd* $\mathcal{G}$) *t* $\Longrightarrow$ (*s*, *t*) $\in$ *gmctxt-cl UNIV* (*agtt-lang* $\mathcal{G}$)
by (*auto simp*: *agtt-lang-def*)

definition *gtt-accept* **where**
gtt-accept $\mathcal{G}$ *s t* $\equiv$ (*s*, *t*) $\in$ *gmctxt-cl UNIV* (*agtt-lang* $\mathcal{G}$)

lemma *gtt-accept-intros*:
(*s*, *t*) $\in$ *agtt-lang* $\mathcal{G}$ $\Longrightarrow$ *gtt-accept* $\mathcal{G}$ *s t*
length ss = *length ts* $\Longrightarrow$ $\forall$ *i* < *length ts*. *gtt-accept* $\mathcal{G}$ (*ss* ! *i*) (*ts* ! *i*) $\Longrightarrow$
(*f*, *length ss*) $\in$ $\mathcal{F}$ $\Longrightarrow$ *gtt-accept* $\mathcal{G}$ (*GFun f ss*) (*GFun f ts*)
by (*auto simp*: *gtt-accept-def*)

abbreviation *gtt-lang-terms* :: ('*q*, '*f*) *gtt* $\Rightarrow$ ('*f*, '*q*) *term rel* **where**
gtt-lang-terms $\mathcal{G}$ $\equiv$ (λ *s*. *map-both term-of-gterm s*) ‘ (*gmctxt-cl UNIV* (*agtt-lang* $\mathcal{G}$))

lemma *term-of-gterm-gtt-lang-gtt-lang-terms-conv*:
map-both term-of-gterm ‘ *gtt-lang* $\mathcal{G}$ = *gtt-lang-terms* $\mathcal{G}$
by *auto*

lemma *gtt-accept-swap* [*simp*]:
gtt-accept (*prod.swap* $\mathcal{G}$) *s t* $\longleftrightarrow$ *gtt-accept* $\mathcal{G}$ *t s*
by (*auto simp*: *gmctxt-cl-swap agtt-lang-swap gtt-accept-def*)

lemma *gtt-lang-swap*:
(*gtt-lang* (*A*, *B*))$^{-1}$ = *gtt-lang* (*B*, *A*)
using *gtt-accept-swap*[*of* (*A*, *B*)]
by (*auto simp*: *gtt-accept-def*)

lemma *gtt-accept-exI*:
assumes *gtt-accept* $\mathcal{G}$ *s t*
shows $\exists$ *u*. *u* |∈| *ta-der'* (*fst* $\mathcal{G}$) (*term-of-gterm s*) ∧ *u* |∈| *ta-der'* (*snd* $\mathcal{G}$) (*term-of-gterm t*)
using *assms* **unfolding** *gtt-accept-def*
proof (*induction*)
case (*base s t*)
then show *?case* **unfolding** *agtt-lang-def*
by (*auto simp*: *gta-der-def ta-der-to-ta-der'*)
next
case (*step ss ts f*)
then have *inner*: ∃ *us*. *length us* = *length ss* ∧
(∀ *i* < *length ss*. (*us* ! *i*) |∈| *ta-der'* (*fst* $\mathcal{G}$) (*term-of-gterm* (*ss* ! *i*)) ∧
(*us* ! *i*) |∈| *ta-der'* (*snd* $\mathcal{G}$) (*term-of-gterm* (*ts* ! *i*)))
using *Ex-list-of-length-P*[*of length ss* λ *u i*. *u* |∈| *ta-der'* (*fst* $\mathcal{G}$) (*term-of-gterm* (*ss* ! *i*)) ∧
u |∈| *ta-der'* (*snd* $\mathcal{G}$) (*term-of-gterm* (*ts* ! *i*))]
by *auto*
then obtain *us* **where** *length us* = *length ss* ∧ (∀ *i* < *length ss*.
(*us* ! *i*) |∈| *ta-der'* (*fst* $\mathcal{G}$) (*term-of-gterm* (*ss* ! *i*)) ∧ (*us* ! *i*) |∈| *ta-der'* (*snd* $\mathcal{G}$) (*term-of-gterm* (*ts* ! *i*)))
by *blast*
then have *Fun f us* |∈| *ta-der'* (*fst* $\mathcal{G}$) (*Fun f* (*map term-of-gterm ss*)) ∧
Fun f us |∈| *ta-der'* (*snd* $\mathcal{G}$) (*Fun f* (*map term-of-gterm ts*)) **using** *step*(*1*)
by *fastforce*
then show *?case* **by** (*metis term-of-gterm.simps*)
qed

lemma *agtt-lang-mono*:
assumes *rules* (*fst* $\mathcal{G}$) |⊆| *rules* (*fst* $\mathcal{G}'$) *eps* (*fst* $\mathcal{G}$) |⊆| *eps* (*fst* $\mathcal{G}'$)
rules (*snd* $\mathcal{G}$) |⊆| *rules* (*snd* $\mathcal{G}'$) *eps* (*snd* $\mathcal{G}$) |⊆| *eps* (*snd* $\mathcal{G}'$)
shows *agtt-lang* $\mathcal{G}$ ⊆ *agtt-lang* $\mathcal{G}'$
using *fsubsetD*[*OF ta-der-mono*[*OF assms*(*1*, *2*)]] *ta-der-mono*[*OF assms*(*3*, *4*)]
by (*auto simp*: *agtt-lang-def gta-der-def dest*!: *fsubsetD*[*OF ta-der-mono*[*OF assms*(*1*, *2*)]] *fsubsetD*[*OF ta-der-mono*[*OF assms*(*3*, *4*)]])

lemma *gtt-lang-mono*:

assumes *rules* (*fst* $\mathcal{G}$) |⊆| *rules* (*fst* $\mathcal{G}'$) *eps* (*fst* $\mathcal{G}$) |⊆| *eps* (*fst* $\mathcal{G}'$)
rules (*snd* $\mathcal{G}$) |⊆| *rules* (*snd* $\mathcal{G}'$) *eps* (*snd* $\mathcal{G}$) |⊆| *eps* (*snd* $\mathcal{G}'$)
shows *gtt-lang* $\mathcal{G}$ ⊆ *gtt-lang* $\mathcal{G}'$
using *agtt-lang-mono*[*OF assms*]
by (*intro gmctxt-cl-mono-rel*) *auto*

definition *fmap-states-gtt* **where**
fmap-states-gtt f ≡ *map-both* (*fmap-states-ta f*)

lemma *ground-map-vars-term-simp*:
ground t ⟹ *map-term f g t* = *map-term f* (λ-. *undefined*) *t*
by (*induct t*) *auto*

lemma *states-fmap-states-gtt* [*simp*]:
gtt-states (*fmap-states-gtt f* $\mathcal{G}$) = *f* |‘| *gtt-states* $\mathcal{G}$
by (*simp add*: *fimage-funion gtt-states-def fmap-states-gtt-def*)

lemma *agtt-lang-fmap-states-gtt*:
assumes *finj-on f* (*gtt-states* $\mathcal{G}$)
shows *agtt-lang* (*fmap-states-gtt f* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
proof −
from *assms* **have** *inj*: *finj-on f* ($\mathcal{Q}$ (*fst* $\mathcal{G}$) |∪| $\mathcal{Q}$ (*snd* $\mathcal{G}$)) *finj-on f* ($\mathcal{Q}$ (*fst* $\mathcal{G}$))
finj-on f ($\mathcal{Q}$ (*snd* $\mathcal{G}$))
by (*auto simp*: *gtt-states-def finj-on-fUn*)
then have *?Ls* ⊆ *?Rs* **using** *ta-der-fmap-states-inv-superset*[*OF - inj*(*1*)]
by (*auto simp*: *agtt-lang-def gta-der-def fmap-states-gtt-def*)
moreover have *?Rs* ⊆ *?Ls*
by (*auto simp*: *agtt-lang-def gta-der-def fmap-states-gtt-def elim*!: *ta-der-to-fmap-states-der*)
ultimately show *?thesis* **by** *blast*
qed

lemma *agtt-lang-Inl-Inr-states-agtt*:
agtt-lang (*fmap-states-gtt Inl* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
agtt-lang (*fmap-states-gtt Inr* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
by (*auto simp*: *finj-Inl-Inr intro*!: *agtt-lang-fmap-states-gtt*)

lemma *gtt-lang-fmap-states-gtt*:
assumes *finj-on f* (*gtt-states* $\mathcal{G}$)
shows *gtt-lang* (*fmap-states-gtt f* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
unfolding *fmap-states-gtt-def*
using *agtt-lang-fmap-states-gtt*[*OF assms*]
by (*simp add*: *fmap-states-gtt-def*)

definition *gtt-only-reach* **where**
gtt-only-reach = *map-both ta-only-reach*

4.1 (A)GTT reachable states

lemma *agtt-only-reach-lang*:

agtt-lang (*gtt-only-reach* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
unfolding *agtt-lang-def gtt-only-reach-def*
by (*auto simp*: *gta-der-def simp flip*: *ta-der-gterm-only-reach*)

lemma *gtt-only-reach-lang*:
gtt-lang (*gtt-only-reach* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
by (*auto simp*: *agtt-only-reach-lang*)

lemma *gtt-only-reach-syms*:
gtt-syms (*gtt-only-reach* $\mathcal{G}$) |⊆| *gtt-syms* $\mathcal{G}$
by (*auto simp*: *gtt-only-reach-def ta-restrict-def ta-sig-def*)

4.2 (A)GTT productive states

definition *gtt-only-prod* **where**
gtt-only-prod $\mathcal{G}$ = (*let iface* = *gtt-interface* $\mathcal{G}$ *in*
map-both (*ta-only-prod iface*) $\mathcal{G}$)

lemma *agtt-only-prod-lang*:
agtt-lang (*gtt-only-prod* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
proof −
let *?A* = *fst* $\mathcal{G}$ **let** *?B* = *snd* $\mathcal{G}$
have *?Ls* ⊆ *?Rs* **unfolding** *agtt-lang-def gtt-only-prod-def*
by (*auto simp*: *Let-def gta-der-def dest*: *ta-der-ta-only-prod-ta-der*)
moreover
{fix *s t* **assume** (*s*, *t*) ∈ *?Rs*
then obtain *q* **where** *r*: *q* |∈| *ta-der* (*fst* $\mathcal{G}$) (*term-of-gterm s*) *q* |∈| *ta-der* (*snd* $\mathcal{G}$) (*term-of-gterm t*)
by (*auto simp*: *agtt-lang-def gta-der-def*)
then have *q* |∈| *gtt-interface* $\mathcal{G}$ **by** (*auto simp*: *gtt-interface-def*)
then have (*s*, *t*) ∈ *?Ls* **using** *r*
by (*auto simp*: *agtt-lang-def gta-der-def gtt-only-prod-def Let-def intro*!: *exI*[*of* - *q*] *ta-der-only-prod ta-productive-setI*)**}**
ultimately show *?thesis* **by** *auto*
qed

lemma *gtt-only-prod-lang*:
gtt-lang (*gtt-only-prod* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
by (*auto simp*: *agtt-only-prod-lang*)

lemma *gtt-only-prod-syms*:
gtt-syms (*gtt-only-prod* $\mathcal{G}$) |⊆| *gtt-syms* $\mathcal{G}$
by (*auto simp*: *gtt-only-prod-def ta-restrict-def ta-sig-def Let-def*)

4.3 (A)GTT trimming

definition *trim-gtt* **where**
trim-gtt = *gtt-only-prod* ∘ *gtt-only-reach*

lemma *trim-agtt-lang*:

agtt-lang (*trim-gtt G*) = *agtt-lang G*
unfolding *trim-gtt-def comp-def agtt-only-prod-lang agtt-only-reach-lang* **..**

lemma *trim-gtt-lang*:
gtt-lang (*trim-gtt G*) = *gtt-lang G*
unfolding *trim-gtt-def comp-def gtt-only-prod-lang gtt-only-reach-lang* **..**

lemma *trim-gtt-prod-syms*:
gtt-syms (*trim-gtt G*) |⊆| *gtt-syms G*
unfolding *trim-gtt-def* **using** *fsubset-trans*[*OF gtt-only-prod-syms gtt-only-reach-syms*]
by *simp*

4.4 root-cleanliness

A GTT is root-clean if none of its interface states can occur in a non-root positions in the accepting derivations corresponding to its anchored GTT relation.

definition *ta-nr-states* :: ('*q*, '*f*) *ta* ⇒ '*q fset* **where**
ta-nr-states A = |⋃| ((*fset-of-list* ∘ *r-lhs-states*) |`| (*rules A*))

definition *gtt-nr-states* **where**
gtt-nr-states G = *ta-nr-states* (*fst G*) |∪| *ta-nr-states* (*snd G*)

definition *gtt-root-clean* **where**
gtt-root-clean G ⟷ *gtt-nr-states G* |∩| *gtt-interface G* = {||}

4.5 Relabeling

definition *relabel-gtt* :: ('*q* :: *linorder*, '*f*) *gtt* ⇒ (*nat*, '*f*) *gtt* **where**
relabel-gtt G = *fmap-states-gtt* (*map-fset-to-nat* (*gtt-states G*)) *G*

lemma *relabel-agtt-lang* [*simp*]:
agtt-lang (*relabel-gtt G*) = *agtt-lang G*
by (*simp add*: *agtt-lang-fmap-states-gtt map-fset-to-nat-inj relabel-gtt-def*)

lemma *agtt-lang-sig*:
fset (*gtt-syms G*) ⊆ $\mathcal{F}$ ⟹ *agtt-lang G* ⊆ $\mathcal{T}_G$ $\mathcal{F}$ × $\mathcal{T}_G$ $\mathcal{F}$
by (*auto simp*: *agtt-lang-def gta-der-def* $\mathcal{T}_G$*-equivalent-def*)
(*metis ffunas-gterm.rep-eq less-eq-fset.rep-eq subset-iff ta-der-gterm-sig*)+

4.6 epsilon free GTTs

lemma *agtt-lang-gtt-eps-free* [*simp*]:
agtt-lang (*gtt-eps-free* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
by (*auto simp*: *agtt-lang-def gta-der-def gtt-eps-free-def ta-res-eps-free*)

lemma *gtt-lang-gtt-eps-free* [*simp*]:
gtt-lang (*gtt-eps-free* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
by *auto*

end
theory *GTT-Compose*
imports *GTT*
begin

4.7 GTT closure under composition

inductive-set Δ_ε*-set* :: $('q, 'f)$ *ta* $\Rightarrow$ $('q, 'f)$ *ta* $\Rightarrow$ $('q \times 'q)$ *set* **for** $\mathcal{A}$ $\mathcal{B}$ **where**
Δ_ε*-set-cong*: *TA-rule f ps p* $|\in|$ *rules* $\mathcal{A} \Longrightarrow$ *TA-rule f qs q* $|\in|$ *rules* $\mathcal{B} \Longrightarrow$ *length ps* = *length qs* $\Longrightarrow$
$(\bigwedge i.\ i < \textit{length qs} \Longrightarrow (\textit{ps} ! i, \textit{qs} ! i) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B}) \Longrightarrow (p, q) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B}$
| Δ_ε*-set-eps1*: (p, p') $|\in|$ *eps* $\mathcal{A} \Longrightarrow (p, q) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p', q) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B}$
| Δ_ε*-set-eps2*: (q, q') $|\in|$ *eps* $\mathcal{B} \Longrightarrow (p, q) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p, q') \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B}$

lemma Δ_ε*-states*: $\Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B} \subseteq \textit{fset}\ (\mathcal{Q}\ \mathcal{A}\ |\times|\ \mathcal{Q}\ \mathcal{B})$
proof −
{fix p q **assume** $(p, q) \in \Delta_\varepsilon\textit{-set}\ \mathcal{A}\ \mathcal{B}$ **then have** $(p, q) \in \textit{fset}\ (\mathcal{Q}\ \mathcal{A}\ |\times|\ \mathcal{Q}\ \mathcal{B})$
by (*induct*) (*auto dest*: *rule-statesD eps-statesD*)**}**
then show *?thesis* **by** *auto*
qed

lemma *finite-*Δ_ε [*simp*]: *finite* (Δ_ε*-set* $\mathcal{A}$ $\mathcal{B}$)
using *finite-subset*[*OF* Δ_ε*-states*]
by *simp*

context
includes *fset.lifting*
begin
lift-definition Δ_ε :: $('q, 'f)$ *ta* $\Rightarrow$ $('q, 'f)$ *ta* $\Rightarrow$ $('q \times 'q)$ *fset* **is** Δ_ε*-set* **by** *simp*
lemmas Δ_ε*-cong* = Δ_ε*-set-cong* [*Transfer.transferred*]
lemmas Δ_ε*-eps1* = Δ_ε*-set-eps1* [*Transfer.transferred*]
lemmas Δ_ε*-eps2* = Δ_ε*-set-eps2* [*Transfer.transferred*]
lemmas Δ_ε*-cases* = Δ_ε*-set.cases*[*Transfer.transferred*]
lemmas Δ_ε*-induct* [*consumes 1*, *case-names* Δ_ε*-cong* Δ_ε*-eps1* Δ_ε*-eps2*] = Δ_ε*-set.induct*[*Transfer.transferred*
lemmas Δ_ε*-intros* = Δ_ε*-set.intros*[*Transfer.transferred*]
lemmas Δ_ε*-simps* = Δ_ε*-set.simps*[*Transfer.transferred*]
end

lemma *finite-alt-def* [*simp*]:
finite $\{(\alpha, \beta).\ (\exists\, t.\ \textit{ground}\ t \wedge \alpha\ |\in|\ \textit{ta-der}\ \mathcal{A}\ t \wedge \beta\ |\in|\ \textit{ta-der}\ \mathcal{B}\ t)\}$ (**is** *finite ?S*)
by (*auto dest*: *ground-ta-der-states*[*THEN fsubsetD*]
intro!: *finite-subset*[*of ?S fset* ($\mathcal{Q}\ \mathcal{A}\ |\times|\ \mathcal{Q}\ \mathcal{B}$)])

lemma Δ_ε*-def'*:
$\Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} = \{|(\alpha, \beta).\ (\exists\, t.\ \textit{ground}\ t \wedge \alpha\ |\in|\ \textit{ta-der}\ \mathcal{A}\ t \wedge \beta\ |\in|\ \textit{ta-der}\ \mathcal{B}\ t)|\}$
proof (*intro fset-eqI iffI*, *goal-cases lr rl*)
case (*lr x*) **obtain** p q **where** x [*simp*]: $x = (p, q)$ **by** (*cases x*)
have $\exists\, t.\ \textit{ground}\ t \wedge p\ |\in|\ \textit{ta-der}\ \mathcal{A}\ t \wedge q\ |\in|\ \textit{ta-der}\ \mathcal{B}\ t$ **using** *lr* **unfolding** x

proof (*induct rule*: Δ_ε-*induct*)
case (Δ_ε-*cong f ps p qs q*)
obtain *ts* **where** *ts*: *ground* (*ts i*) $\wedge$ *ps* ! *i* $|\in|$ *ta-der* $\mathcal{A}$ (*ts i*) $\wedge$ *qs* ! *i* $|\in|$ *ta-der* $\mathcal{B}$ (*ts i*)
if *i* < *length qs* **for** *i* **using** Δ_ε-*cong*(*5*) **by** *metis*
then show *?case* **using** Δ_ε-*cong*(*1*−*3*)
by (*auto intro*!: *exI*[*of* - *Fun f* (*map ts* [*0*..<*length qs*])]) *blast*+
qed (*meson ta-der-eps*)+
then show *?case* **by** *auto*
next
case (*rl x*) **obtain** *p q* **where** *x* [*simp*]: *x* = (*p*, *q*) **by** (*cases x*)
obtain *t* **where** *ground t p* $|\in|$ *ta-der* $\mathcal{A}$ *t q* $|\in|$ *ta-der* $\mathcal{B}$ *t* **using** *rl* **by** *auto*
then show *?case* **unfolding** *x*
proof (*induct t arbitrary*: *p q*)
case (*Fun f ts*)
obtain *p*′ *ps* **where** *p*′: *TA-rule f ps p*′ $|\in|$ *rules* $\mathcal{A}$ *p*′ = *p* $\vee$ (*p*′, *p*) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ *length ps* = *length ts*
$\bigwedge i.\ i < \text{length } ts \Longrightarrow ps\ !\ i\ |\in|$ *ta-der* $\mathcal{A}$ (*ts* ! *i*) **using** *Fun*(*3*) **by** *auto*
obtain *q*′ *qs* **where** *q*′: *f qs* $\to$ *q*′ $|\in|$ *rules* $\mathcal{B}$ *q*′ = *q* $\vee$ (*q*′, *q*) $|\in|$ (*eps* $\mathcal{B}$)$|^+|$ *length qs* = *length ts*
$\bigwedge i.\ i < \text{length } ts \Longrightarrow qs\ !\ i\ |\in|$ *ta-der* $\mathcal{B}$ (*ts* ! *i*) **using** *Fun*(*4*) **by** *auto*
have *st*: (*p*′, *q*′) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$
using *Fun*(*1*)[*OF nth-mem* - *p*′(*4*) *q*′(*4*)] *Fun*(*2*) *p*′(*3*) *q*′(*3*)
by (*intro* Δ_ε-*cong*[*OF p*′(*1*) *q*′(*1*)]) *auto*
{assume (*p*′, *p*) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ **then have** (*p*, *q*′) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ **using** *st*
by (*induct rule*: *ftrancl-induct*) (*auto intro*: Δ_ε-*eps1*)**}**
from *st this p*′(*2*) **have** *st*: (*p*, *q*′) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ **by** *auto*
{assume (*q*′, *q*) $|\in|$ (*eps* $\mathcal{B}$)$|^+|$ **then have** (*p*, *q*) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ **using** *st*
by (*induct rule*: *ftrancl-induct*) (*auto intro*: Δ_ε-*eps2*)**}**
from *st this q*′(*2*) **show** (*p*, *q*) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ **by** *auto*
qed *auto*
qed

lemma Δ_ε-*fmember*:
(*p*, *q*) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ $\longleftrightarrow$ ($\exists t.$ *ground t* $\wedge$ *p* $|\in|$ *ta-der* $\mathcal{A}$ *t* $\wedge$ *q* $|\in|$ *ta-der* $\mathcal{B}$ *t*)
by (*auto simp*: Δ_ε-*def*′)

definition *GTT-comp* :: ('*q*, '*f*) *gtt* $\Rightarrow$ ('*q*, '*f*) *gtt* $\Rightarrow$ ('*q*, '*f*) *gtt* **where**
GTT-comp $\mathcal{G}_1$ $\mathcal{G}_2$ =
(*let* Δ = Δ_ε (*snd* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$) *in*
(*TA* (*gtt-rules* (*fst* $\mathcal{G}_1$, *fst* $\mathcal{G}_2$)) (*eps* (*fst* $\mathcal{G}_1$) $|\cup|$ *eps* (*fst* $\mathcal{G}_2$) $|\cup|$ Δ),
TA (*gtt-rules* (*snd* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$)) (*eps* (*snd* $\mathcal{G}_1$) $|\cup|$ *eps* (*snd* $\mathcal{G}_2$) $|\cup|$ ($\Delta|^{-1}|$))))

lemma *gtt-syms-GTT-comp*:
gtt-syms (*GTT-comp A B*) = *gtt-syms A* $|\cup|$ *gtt-syms B*
by (*auto simp*: *GTT-comp-def ta-sig-def Let-def*)

lemma Δ_ε-*statesD*:
(*p*, *q*) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B}$ $\Longrightarrow$ *p* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$

$(p,\ q)\ |\in|\ \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} \Longrightarrow q\ |\in|\ \mathcal{Q}\ \mathcal{B}$
using *subsetD*[*OF* Δ_ε-*states*, *of* $(p,\ q)\ \mathcal{A}\ \mathcal{B}$]
by (*auto simp flip*: Δ_ε.*rep-eq*)

lemma Δ_ε-*statesD'*:
$q\ |\in|$ *eps-states* $(\Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}) \Longrightarrow q\ |\in|\ \mathcal{Q}\ \mathcal{A}\ |\cup|\ \mathcal{Q}\ \mathcal{B}$
by (*auto simp*: *eps-states-def dest*: Δ_ε-*statesD*)

lemma Δ_ε-*swap*:
prod.swap $p\ |\in|\ \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} \longleftrightarrow p\ |\in|\ \Delta_\varepsilon\ \mathcal{B}\ \mathcal{A}$
by (*auto simp*: Δ_ε-*def'*)

lemma Δ_ε-*inverse* [*simp*]:
$(\Delta_\varepsilon\ \mathcal{A}\ \mathcal{B})|^{-1}| = \Delta_\varepsilon\ \mathcal{B}\ \mathcal{A}$
by (*auto simp*: Δ_ε-*def'*)

lemma *gtt-states-comp-union*:
gtt-states $(GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)\ |\subseteq|$ *gtt-states* $\mathcal{G}_1\ |\cup|$ *gtt-states* $\mathcal{G}_2$
proof (*intro fsubsetI*, *goal-cases lr*)
case (*lr q*) **then show** *?case*
by (*auto simp*: *GTT-comp-def gtt-states-def* $\mathcal{Q}$-*def dest*: Δ_ε-*statesD'*)
qed

lemma *GTT-comp-swap* [*simp*]:
GTT-comp (*prod.swap* $\mathcal{G}_2$) (*prod.swap* $\mathcal{G}_1$) = *prod.swap* (*GTT-comp* $\mathcal{G}_1\ \mathcal{G}_2$)
by (*simp add*: *GTT-comp-def ac-simps*)

lemma *gtt-comp-complete-semi*:
assumes *s*: $q\ |\in|$ *gta-der* (*fst* $\mathcal{G}_1$) s **and** *u*: $q\ |\in|$ *gta-der* (*snd* $\mathcal{G}_1$) u **and** *ut*: *gtt-accept* $\mathcal{G}_2\ u\ t$
shows $q\ |\in|$ *gta-der* (*fst* (*GTT-comp* $\mathcal{G}_1\ \mathcal{G}_2$)) $s\ q\ |\in|$ *gta-der* (*snd* (*GTT-comp* $\mathcal{G}_1\ \mathcal{G}_2$)) t
proof (*goal-cases L R*)
let *?*$\mathcal{G}$ = *GTT-comp* $\mathcal{G}_1\ \mathcal{G}_2$
have *sub1l*: *rules* (*fst* $\mathcal{G}_1$) $|\subseteq|$ *rules* (*fst ?*$\mathcal{G}$) *eps* (*fst* $\mathcal{G}_1$) $|\subseteq|$ *eps* (*fst ?*$\mathcal{G}$)
and *sub1r*: *rules* (*snd* $\mathcal{G}_1$) $|\subseteq|$ *rules* (*snd ?*$\mathcal{G}$) *eps* (*snd* $\mathcal{G}_1$) $|\subseteq|$ *eps* (*snd ?*$\mathcal{G}$)
and *sub2r*: *rules* (*snd* $\mathcal{G}_2$) $|\subseteq|$ *rules* (*snd ?*$\mathcal{G}$) *eps* (*snd* $\mathcal{G}_2$) $|\subseteq|$ *eps* (*snd ?*$\mathcal{G}$)
by (*auto simp*: *GTT-comp-def*)
{ case *L* **then show** *?case* **using** *s ta-der-mono*[*OF sub1l*]
by (*auto simp*: *gta-der-def*)
next
case *R* **then show** *?case* **using** *ut u* **unfolding** *gtt-accept-def*
proof (*induct arbitrary*: *q s*)
case (*base s t*)
from *base*(*1*) **obtain** *p* **where** *p*: $p\ |\in|$ *gta-der* (*fst* $\mathcal{G}_2$) $s\ p\ |\in|$ *gta-der* (*snd* $\mathcal{G}_2$) t
by (*auto simp*: *agtt-lang-def*)
then have $(p,\ q)\ |\in|$ *eps* (*snd* (*GTT-comp* $\mathcal{G}_1\ \mathcal{G}_2$))

```
      using Δε-fmember[of p q fst G2 snd G1] base(2)
      by (auto simp: GTT-comp-def gta-der-def)
    from ta-der-eps[OF this] show ?case using p ta-der-mono[OF sub2r]
      by (auto simp add: gta-der-def)
  next
    case (step ss ts f)
    from step(1, 4) obtain ps p where TA-rule f ps p |∈| rules (snd G1) p = q
∨ (p, q) |∈| (eps (snd G1))|+|
      length ps = length ts ⋀i. i < length ts ⟹ ps ! i |∈| gta-der (snd G1) (ss !
i)
      unfolding gta-der-def by auto
    then show ?case using step(1, 2) sub1r(1) ftrancl-mono[OF - sub1r(2)]
      by (auto simp: gta-der-def intro!: exI[of - p] exI[of - ps])
  qed}
qed

lemmas gtt-comp-complete-semi' = gtt-comp-complete-semi[of - prod.swap G2 - -
prod.swap G1 for G1 G2,
  unfolded fst-swap snd-swap GTT-comp-swap gtt-accept-swap]

lemma gtt-comp-acomplete:
  gcomp-rel UNIV (agtt-lang G1) (agtt-lang G2) ⊆ agtt-lang (GTT-comp G1 G2)
proof (intro subrelI, goal-cases LR)
  case (LR s t)
  then consider
      q u where q |∈| gta-der (fst G1) s q |∈| gta-der (snd G1) u gtt-accept G2 u t
    | q u where q |∈| gta-der (snd G2) t q |∈| gta-der (fst G2) u gtt-accept G1 s u
    by (auto simp: gcomp-rel-def gtt-accept-def elim!: agtt-langE)
  then show ?case
  proof (cases)
    case 1 show ?thesis using gtt-comp-complete-semi[OF 1]
      by (auto simp: agtt-lang-def gta-der-def)
  next
    case 2 show ?thesis using gtt-comp-complete-semi'[OF 2]
      by (auto simp: agtt-lang-def gta-der-def)
  qed
qed

lemma Δε-steps-from-G2:
  assumes (q, q') |∈| (eps (fst (GTT-comp G1 G2)))|+| q |∈| gtt-states G2
    gtt-states G1 |∩| gtt-states G2 = {||}
  shows (q, q') |∈| (eps (fst G2))|+| ∧ q' |∈| gtt-states G2
  using assms(1−2)
proof (induct rule: converse-ftrancl-induct)
  case (Base y)
  then show ?case using assms(3)
   by (fastforce simp: GTT-comp-def gtt-states-def dest: eps-statesD Δε-statesD(1))
next
  case (Step q p)
```

have $(q,\ p)\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+|\ p\ |\in|\ gtt\text{-}states\ \mathcal{G}_2$
using *Step*(*1*, *4*) *assms*(*3*)
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *eps-statesD* Δ_ε*-statesD*(*1*))
then show *?case* **using** *Step*(*3*)
by (*auto intro*: *ftrancl-trans*)
qed

lemma Δ_ε*-steps-from-*$\mathcal{G}_1$:
assumes $(p,\ r)\ |\in|\ (eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|\ p\ |\in|\ gtt\text{-}states\ \mathcal{G}_1$
$gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
obtains $r\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ (p,\ r)\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$|\ q\ p'$ **where** $r\ |\in|\ gtt\text{-}states\ \mathcal{G}_2\ p = p' \vee (p,\ p')\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|\ (p',\ q)\ |\in|$ $\Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2)$
$q = r \vee (q,\ r)\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+|$
using *assms*(*1*,*2*)
proof (*induct arbitrary*: *thesis rule*: *converse-ftrancl-induct*)
case (*Base p*)
from *Base*(*1*) **consider** (*a*) $(p,\ r)\ |\in|\ eps\ (fst\ \mathcal{G}_1)\ |$ (*b*) $(p,\ r)\ |\in|\ eps\ (fst\ \mathcal{G}_2)\ |$
(*c*) $(p,\ r)\ |\in|\ (\Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2))$
by (*auto simp*: *GTT-comp-def*)
then show *?case* **using** *assms*(*3*) *Base*
by *cases* (*auto simp*: *GTT-comp-def gtt-states-def dest*: *eps-statesD* Δ_ε*-statesD*)
next
case (*Step q p*)
consider $(q,\ p)\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|\ p\ |\in|\ gtt\text{-}states\ \mathcal{G}_1$
$|\ (q,\ p)\ |\in|\ \Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2)\ p\ |\in|\ gtt\text{-}states\ \mathcal{G}_2$ **using** *assms*(*3*) *Step*(*1*, *6*)
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *eps-statesD* Δ_ε*-statesD*)
then show *?case*
proof (*cases*)
case *1* **note** $a = 1$ **show** *?thesis*
proof (*cases rule*: *Step*(*3*))
case (*2 p' q*)
then show *?thesis* **using** *assms a*
by (*auto intro*: *Step*(*5*) *ftrancl-trans*)
qed (*auto simp*: *a*(*2*) *intro*: *Step*(*4*) *ftrancl-trans*[*OF a*(*1*)])
next
case *2* **show** *?thesis* **using** Δ_ε*-steps-from-*$\mathcal{G}_2$[*OF Step*(*2*) *2*(*2*) *assms*(*3*)] *Step*(*5*)[*OF - - 2*(*1*)] **by** *auto*
qed
qed

lemma Δ_ε*-steps-from-*$\mathcal{G}_1$*-*$\mathcal{G}_2$:
assumes $(q,\ q')\ |\in|\ (eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|$ $gtt\text{-}states\ \mathcal{G}_2$
$gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
obtains $q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ q'\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ (q,\ q')\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$|\ p\ p'$ **where** $q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ q'\ |\in|\ gtt\text{-}states\ \mathcal{G}_2\ q = p \vee (q,\ p)\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$(p,\ p')\ |\in|\ \Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2)\ p' = q' \vee (p',\ q')\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+|$

| q |∈| *gtt-states* $\mathcal{G}_2$ (q, q') |∈| $(eps\ (fst\ \mathcal{G}_2))|^+|$ ∧ q' |∈| *gtt-states* $\mathcal{G}_2$
using *assms* Δ_ε*-steps-from-*$\mathcal{G}_1$ Δ_ε*-steps-from-*$\mathcal{G}_2$
by (*metis funion-iff*)

lemma *GTT-comp-eps-fst-statesD*:
(p, q) |∈| *eps* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) ⟹ p |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$
(p, q) |∈| *eps* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) ⟹ q |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *eps-statesD* Δ_ε*-statesD*)

lemma *GTT-comp-eps-ftrancl-fst-statesD*:
(p, q) |∈| $(eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|$ ⟹ p |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$
(p, q) |∈| $(eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|$ ⟹ q |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$
using *GTT-comp-eps-fst-statesD*[*of* - - $\mathcal{G}_1$ $\mathcal{G}_2$]
by (*meson converse-ftranclE ftranclE*)+

lemma *GTT-comp-first*:
assumes q |∈| *ta-der* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) t q |∈| *gtt-states* $\mathcal{G}_1$
gtt-states $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows q |∈| *ta-der* (*fst* $\mathcal{G}_1$) t
using *assms*(*1*,*2*)
proof (*induct t arbitrary*: q)
case (*Var* q')
have $q \neq q' \Longrightarrow q'$ |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$ **using** *Var*
by (*auto dest*: *GTT-comp-eps-ftrancl-fst-statesD*)
then show *?case* **using** *Var assms*(*3*)
by (*auto elim*: Δ_ε*-steps-from-*$\mathcal{G}_1$*-*$\mathcal{G}_2$)
next
case (*Fun f ts*)
obtain q' *qs* **where** q': *TA-rule f qs* q' |∈| *rules* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$))
$q' = q \vee (q', q)$ |∈| $(eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|$ *length qs* = *length ts*
⋀i. $i <$ *length ts* ⟹ *qs* ! i |∈| *ta-der* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) (*ts* ! i)
using *Fun*(*2*) **by** *auto*
have q' |∈| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$ **using** q'(*1*)
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *rule-statesD*)
then have *st*: q' |∈| *gtt-states* $\mathcal{G}_1$ **and** *eps*:$q' = q \vee (q', q)$ |∈| $(eps\ (fst\ \mathcal{G}_1))|^+|$
using q'(*2*) *Fun*(*3*) *assms*(*3*)
by (*auto elim*!: Δ_ε*-steps-from-*$\mathcal{G}_1$*-*$\mathcal{G}_2$)
from *st* **have** *rule*: *TA-rule f qs* q' |∈| *rules* (*fst* $\mathcal{G}_1$) **using** *assms*(*3*) q'(*1*)
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *rule-statesD*)
have $i <$ *length ts* ⟹ *qs* ! i |∈| *ta-der* (*fst* $\mathcal{G}_1$) (*ts* ! i) **for** i
using *rule* q'(*3*, *4*)
by (*intro Fun*(*1*)[*OF nth-mem*]) (*auto simp*: *gtt-states-def dest*!: *rule-statesD*(*4*))
then show *?case* **using** q'(*3*) *rule eps*
by *auto*
qed

lemma *GTT-comp-second*:

assumes *gtt-states* $\mathcal{G}_1$ $|\cap|$ *gtt-states* $\mathcal{G}_2 = \{||\}$ *q* $|\in|$ *gtt-states* $\mathcal{G}_2$
q $|\in|$ *ta-der* (*snd* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) *t*
shows *q* $|\in|$ *ta-der* (*snd* $\mathcal{G}_2$) *t*
using *assms GTT-comp-first*[*of q prod.swap* $\mathcal{G}_2$ *prod.swap* $\mathcal{G}_1$]
by (*auto simp*: *gtt-states-def*)

lemma *gtt-comp-sound-semi*:
fixes $\mathcal{G}_1$ $\mathcal{G}_2$ :: ('*f*, '*q*) *gtt*
assumes *as2*: *gtt-states* $\mathcal{G}_1$ $|\cap|$ *gtt-states* $\mathcal{G}_2 = \{||\}$
and *1*: *q* $|\in|$ *gta-der* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) *s q* $|\in|$ *gta-der* (*snd* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) *t q* $|\in|$ *gtt-states* $\mathcal{G}_1$
shows $\exists$ *u*. *q* $|\in|$ *gta-der* (*snd* $\mathcal{G}_1$) *u* $\wedge$ *gtt-accept* $\mathcal{G}_2$ *u t* **using** *1*(*2*,*3*) **unfolding** *gta-der-def*
proof (*induct rule*: *ta-der-gterm-induct*)
case (*GFun f ts ps p q*)
show *?case*
proof (*cases p* $|\in|$ *gtt-states* $\mathcal{G}_1$)
case *True*
then have $*$: *TA-rule f ps p* $|\in|$ *rules* (*snd* $\mathcal{G}_1$) **using** *GFun*(*1*, *6*) *as2*
by (*auto simp*: *GTT-comp-def gtt-states-def dest*: *rule-statesD*)
moreover have *st*: *i* < *length ps* $\Longrightarrow$ *ps* ! *i* $|\in|$ *gtt-states* $\mathcal{G}_1$ **for** *i* **using** $*$
by (*force simp*: *gtt-states-def dest*: *rule-statesD*)
moreover have *i* < *length ps* $\Longrightarrow$ $\exists$ *u*. *ps* ! *i* $|\in|$ *ta-der* (*snd* $\mathcal{G}_1$) (*term-of-gterm u*) $\wedge$ *gtt-accept* $\mathcal{G}_2$ *u* (*ts* ! *i*) **for** *i*
using *st GFun*(*2*) **by** (*intro GFun*(*5*)) *simp*
then obtain *us* **where**
$\bigwedge$*i*. *i* < *length ps* $\Longrightarrow$ *ps* ! *i* $|\in|$ *ta-der* (*snd* $\mathcal{G}_1$) (*term-of-gterm* (*us i*)) $\wedge$ *gtt-accept* $\mathcal{G}_2$ (*us i*) (*ts* ! *i*)
by *metis*
moreover have *p* = *q* $\vee$ (*p*, *q*) $|\in|$ (*eps* (*snd* $\mathcal{G}_1$))$|^+|$ **using** *GFun*(*3*, *6*) *True as2*
by (*auto simp*: *gtt-states-def elim*!: Δ_ε-*steps-from-*$\mathcal{G}_1$-$\mathcal{G}_2$[*of p q prod.swap* $\mathcal{G}_2$ *prod.swap* $\mathcal{G}_1$, *simplified*])
ultimately show *?thesis* **using** *GFun*(*2*)
by (*intro exI*[*of* - *GFun f* (*map us* [*0*..<*length ts*])])
(*auto simp*: *gtt-accept-def intro*!: *exI*[*of* - *ps*] *exI*[*of* - *p*])
next
case *False* **note** *nt-st* = *this*
then have *False*: *p* $\neq$ *q* **using** *GFun*(*6*) **by** *auto*
then have *eps*: (*p*, *q*) $|\in|$ (*eps* (*snd* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)))$|^+|$ **using** *GFun*(*3*) **by** *simp*
show *?thesis* **using** Δ_ε-*steps-from-*$\mathcal{G}_1$-$\mathcal{G}_2$[*of p q prod.swap* $\mathcal{G}_2$ *prod.swap* $\mathcal{G}_1$, *simplified*, *OF eps*]
proof (*cases*, *goal-cases*)
case *1* **then show** *?case* **using** *False GFun*(*3*)
by (*metis GTT-comp-eps-ftrancl-fst-statesD*(*1*) *GTT-comp-swap fst-swap funion-iff*)
next
case *2* **then show** *?case* **using** *as2* **by** (*auto simp*: *gtt-states-def*)

```
    next
      case 3 then show ?case using as2 GFun(6) by (auto simp: gtt-states-def)
    next
      case (4 r p')
     have meet: r |∈| ta-der (snd (GTT-comp 𝒢₁ 𝒢₂)) (Fun f (map term-of-gterm
ts))
        using GFun(1 − 4) 4(3) False
       by (auto simp: GTT-comp-def in-ftrancl-UnI intro!: exI[ of - ps] exI[ of - p])
     then obtain u where wit: ground u p' |∈| ta-der (snd 𝒢₁) u r |∈| ta-der (fst
𝒢₂) u
        using 4(4−) unfolding Δ_ε-def' by blast
      from wit(1, 3) have gtt-accept 𝒢₂ (gterm-of-term u) (GFun f ts)
        using GTT-comp-second[OF as2 - meet] unfolding gtt-accept-def
        by (intro gmctxt-cl.base agtt-langI[of r])
             (auto simp add: gta-der-def gtt-states-def simp del: ta-der-Fun dest:
ground-ta-der-states)
      then show ?case using 4(5) wit(1, 2)
        by (intro exI[of - gterm-of-term u]) (auto simp add: ta-der-trancl-eps)
    next
      case 5
      then show ?case using nt-st as2
        by (simp add: gtt-states-def)
    qed
  qed
qed

lemma gtt-comp-asound:
  assumes gtt-states 𝒢₁ |∩| gtt-states 𝒢₂ = {||}
  shows agtt-lang (GTT-comp 𝒢₁ 𝒢₂) ⊆ gcomp-rel UNIV (agtt-lang 𝒢₁) (agtt-lang
𝒢₂)
proof (intro subrelI, goal-cases LR)
  case (LR s t)
  obtain q where q: q |∈| gta-der (fst (GTT-comp 𝒢₁ 𝒢₂)) s q |∈| gta-der (snd
(GTT-comp 𝒢₁ 𝒢₂)) t
    using LR by (auto simp: agtt-lang-def)
  {
    fix 𝒢₁ 𝒢₂ s t assume as2: gtt-states 𝒢₁ |∩| gtt-states 𝒢₂ = {||}
      and 1: q |∈| ta-der (fst (GTT-comp 𝒢₁ 𝒢₂)) (term-of-gterm s)
        q |∈| ta-der (snd (GTT-comp 𝒢₁ 𝒢₂)) (term-of-gterm t) q |∈| gtt-states 𝒢₁
    note st = GTT-comp-first[OF 1(1,3) as2]
    obtain u where u: q |∈| ta-der (snd 𝒢₁) (term-of-gterm u) gtt-accept 𝒢₂ u t
       using gtt-comp-sound-semi[OF as2 1[folded gta-der-def]] by (auto simp:
gta-der-def)
    have (s, u) ∈ agtt-lang 𝒢₁ using st u(1)
      by (auto simp: agtt-lang-def gta-der-def)
    moreover have (u, t) ∈ gtt-lang 𝒢₂ using u(2)
      by (auto simp: gtt-accept-def)
    ultimately have (s, t) ∈ agtt-lang 𝒢₁ O gmctxt-cl UNIV (agtt-lang 𝒢₂)
      by auto}
```

note *base* = *this*
consider q |∈| *gtt-states* $\mathcal{G}_1$ | q |∈| *gtt-states* $\mathcal{G}_2$ | q |∉| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$ **by** *blast*
then show *?case* **using** q *assms*
proof (*cases*, *goal-cases*)
case *1* **then show** *?case* **using** *base*[*of* $\mathcal{G}_1$ $\mathcal{G}_2$ *s t*]
by (*auto simp*: *gcomp-rel-def gta-der-def*)
next
case *2* **then show** *?case* **using** *base*[*of prod.swap* $\mathcal{G}_2$ *prod.swap* $\mathcal{G}_1$ *t s*, *THEN converseI*]
by (*auto simp*: *gcomp-rel-def converse-relcomp converse-agtt-lang gta-der-def gtt-states-def*)
(*simp add*: *finter-commute funion-commute gtt-lang-swap prod.swap-def*)+
next
case *3* **then show** *?case* **using** *fsubsetD*[*OF gtt-states-comp-union*[*of* $\mathcal{G}_1$ $\mathcal{G}_2$], *of q*]
by (*auto simp*: *gta-der-def gtt-states-def*)
qed
qed

lemma *gtt-comp-lang-complete*:
shows *gtt-lang* $\mathcal{G}_1$ *O gtt-lang* $\mathcal{G}_2$ ⊆ *gtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)
using *gmctxt-cl-mono-rel*[*OF gtt-comp-acomplete*, *of UNIV* $\mathcal{G}_1$ $\mathcal{G}_2$]
by (*simp only*: *gcomp-rel*[*symmetric*])

lemma *gtt-comp-alang*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gcomp-rel UNIV* (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$)
by (*intro equalityI gtt-comp-asound*[*OF assms*] *gtt-comp-acomplete*)

lemma *gtt-comp-lang*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *gtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gtt-lang* $\mathcal{G}_1$ *O gtt-lang* $\mathcal{G}_2$
by (*simp only*: *arg-cong*[*OF gtt-comp-alang*[*OF assms*], *of gmctxt-cl UNIV*] *gcomp-rel*)

abbreviation *GTT-comp′* **where**
GTT-comp′ $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *GTT-comp* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *gtt-comp′-alang*:
shows *agtt-lang* (*GTT-comp′* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gcomp-rel UNIV* (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$)
proof −
have [*simp*]: *finj-on Inl* (*gtt-states* $\mathcal{G}_1$) *finj-on Inr* (*gtt-states* $\mathcal{G}_2$)
by (*auto simp add*: *finj-on.rep-eq*)
then show *?thesis*
by (*subst gtt-comp-alang*) (*auto simp*: *agtt-lang-fmap-states-gtt*)
qed

end
theory *GTT-Transitive-Closure*
imports *GTT-Compose*
begin

4.8 GTT closure under transitivity

inductive-set Δ-*trancl-set* :: ('*q*, '*f*) *ta* ⇒ ('*q*, '*f*) *ta* ⇒ ('*q* × '*q*) *set* **for** *A B* **where**
Δ-*set-cong*: *TA-rule f ps p* |∈| *rules A* ⟹ *TA-rule f qs q* |∈| *rules B* ⟹ *length ps* = *length qs* ⟹
(⋀*i*. *i* < *length qs* ⟹ (*ps* ! *i*, *qs* ! *i*) ∈ Δ-*trancl-set A B*) ⟹ (*p*, *q*) ∈ Δ-*trancl-set A B*
| Δ-*set-eps1*: (*p*, *p*′) |∈| *eps A* ⟹ (*p*, *q*) ∈ Δ-*trancl-set A B* ⟹ (*p*′, *q*) ∈ Δ-*trancl-set A B*
| Δ-*set-eps2*: (*q*, *q*′) |∈| *eps B* ⟹ (*p*, *q*) ∈ Δ-*trancl-set A B* ⟹ (*p*, *q*′) ∈ Δ-*trancl-set A B*
| Δ-*set-trans*: (*p*, *q*) ∈ Δ-*trancl-set A B* ⟹ (*q*, *r*) ∈ Δ-*trancl-set A B* ⟹ (*p*, *r*) ∈ Δ-*trancl-set A B*

lemma Δ-*trancl-set-states*: Δ-*trancl-set* 𝒜 ℬ ⊆ *fset* (𝒬 𝒜 |×| 𝒬 ℬ)
proof −
{fix *p q* **assume** (*p*, *q*) ∈ Δ-*trancl-set* 𝒜 ℬ **then have** (*p*, *q*) ∈ *fset* (𝒬 𝒜 |×| 𝒬 ℬ)
by (*induct*) (*auto dest*: *rule-statesD eps-statesD*)**}**
then show *?thesis* **by** *auto*
qed

lemma *finite*-Δ-*trancl-set* [*simp*]: *finite* (Δ-*trancl-set* 𝒜 ℬ)
using *finite-subset*[*OF* Δ-*trancl-set-states*]
by *simp*

context
includes *fset.lifting*
begin
lift-definition Δ-*trancl* :: ('*q*, '*f*) *ta* ⇒ ('*q*, '*f*) *ta* ⇒ ('*q* × '*q*) *fset* **is** Δ-*trancl-set*
by *simp*
lemmas Δ-*trancl-cong* = Δ-*set-cong* [*Transfer.transferred*]
lemmas Δ-*trancl-eps1* = Δ-*set-eps1* [*Transfer.transferred*]
lemmas Δ-*trancl-eps2* = Δ-*set-eps2* [*Transfer.transferred*]
lemmas Δ-*trancl-cases* = Δ-*trancl-set.cases*[*Transfer.transferred*]
lemmas Δ-*trancl-induct* [*consumes 1*, *case-names* Δ-*cong* Δ-*eps1* Δ-*eps2* Δ-*trans*] = Δ-*trancl-set.induct*[*Transfer.transferred*]
lemmas Δ-*trancl-intros* = Δ-*trancl-set.intros*[*Transfer.transferred*]
lemmas Δ-*trancl-simps* = Δ-*trancl-set.simps*[*Transfer.transferred*]
end

lemma Δ-*trancl-cl* [*simp*]:

```
  (Δ-trancl A B)|^+| = Δ-trancl A B
proof −
  {fix s t assume (s, t) |∈| (Δ-trancl A B)|^+| then have (s, t) |∈| Δ-trancl A B
     by (induct rule: ftrancl-induct) (auto intro: Δ-trancl-intros)}
  then show ?thesis by auto
qed

lemma Δ-trancl-states: Δ-trancl 𝒜 ℬ |⊆| (𝒬 𝒜 |×| 𝒬 ℬ)
  using Δ-trancl-set-states
  by (metis Δ-trancl.rep-eq fSigma-cong less-eq-fset.rep-eq)

definition GTT-trancl where
  GTT-trancl G =
    (let Δ = Δ-trancl (snd G) (fst G) in
    (TA (rules (fst G)) (eps (fst G) |∪| Δ),
                TA (rules (snd G)) (eps (snd G) |∪| (Δ|^-1|))))

lemma Δ-trancl-inv:
  (Δ-trancl A B)|^-1| = Δ-trancl B A
proof −
  have [dest]: (p, q) |∈| Δ-trancl A B ⟹ (q, p) |∈| Δ-trancl B A for p q A B
    by (induct rule: Δ-trancl-induct) (auto intro: Δ-trancl-intros)
  show ?thesis by auto
qed

lemma gtt-states-GTT-trancl:
  gtt-states (GTT-trancl G) |⊆| gtt-states G
  unfolding GTT-trancl-def
 by (auto simp: gtt-states-def 𝒬-def Δ-trancl-inv dest!: fsubsetD[OF Δ-trancl-states])

lemma gtt-syms-GTT-trancl:
  gtt-syms (GTT-trancl G) = gtt-syms G
  by (auto simp: GTT-trancl-def ta-sig-def Δ-trancl-inv)

lemma GTT-trancl-base:
  gtt-lang G ⊆ gtt-lang (GTT-trancl G)
 using gtt-lang-mono[of G GTT-trancl G] by (auto simp: Δ-trancl-inv GTT-trancl-def)

lemma GTT-trancl-trans:
  gtt-lang (GTT-comp (GTT-trancl G) (GTT-trancl G)) ⊆ gtt-lang (GTT-trancl
G)
proof −
  have [dest]: (p, q) |∈| Δ_ε (TA (rules A) (eps A |∪| (Δ-trancl B A)))
    (TA (rules B) (eps B |∪| (Δ-trancl A B))) ⟹ (p, q) |∈| Δ-trancl A B for p q
A B
    by (induct rule: Δ_ε-induct) (auto intro: Δ-trancl-intros simp: Δ-trancl-inv[of
B A, symmetric])
  show ?thesis
   by (intro gtt-lang-mono[of GTT-comp (GTT-trancl G) (GTT-trancl G) GTT-trancl
```

G])
(*auto simp*: *GTT-comp-def GTT-trancl-def* Δ-*trancl-inv*)
qed

lemma *agtt-lang-base*:
agtt-lang G ⊆ *agtt-lang* (*GTT-trancl G*)
by (*rule agtt-lang-mono*) (*auto simp*: *GTT-trancl-def* Δ-*trancl-inv*)

lemma Δ_ε-*tr-incl*:
Δ_ε (*TA* (*rules A*) (*eps A* |∪| Δ-*trancl B A*)) (*TA* (*rules B*) (*eps B* |∪| Δ-*trancl A B*)) = Δ-*trancl A B*
(**is** *?LS* = *?RS*)
proof −
{fix *p q* **assume** (*p*, *q*) |∈| *?LS* **then have** (*p*, *q*) |∈| *?RS*
by (*induct rule*: Δ_ε-*induct*)
(*auto simp*: Δ-*trancl-inv*[*of B A*, *symmetric*] *intro*: Δ-*trancl-intros*)**}**
moreover
{fix *p q* **assume** (*p*, *q*) |∈| *?RS* **then have** (*p*, *q*) |∈| *?LS*
by (*induct rule*: Δ-*trancl-induct*)
(*auto simp*: Δ-*trancl-inv*[*of B A*, *symmetric*] *intro*: Δ_ε-*intros*)**}**
ultimately show *?thesis*
by *auto*
qed

lemma *agtt-lang-trans*:
gcomp-rel UNIV (*agtt-lang* (*GTT-trancl G*)) (*agtt-lang* (*GTT-trancl G*)) ⊆ *agtt-lang* (*GTT-trancl G*)
proof −
have [*intro*!, *dest*]: (*p*, *q*) |∈| Δ_ε (*TA* (*rules A*) (*eps A* |∪| (Δ-*trancl B A*)))
(*TA* (*rules B*) (*eps B* |∪| (Δ-*trancl A B*))) ⟹ (*p*, *q*) |∈| Δ-*trancl A B* **for** *p q A B*
by (*induct rule*: Δ_ε-*induct*) (*auto intro*: Δ-*trancl-intros simp*: Δ-*trancl-inv*[*of B A*, *symmetric*])
show *?thesis*
by (*rule subset-trans*[*OF gtt-comp-acomplete agtt-lang-mono*])
(*auto simp*: *GTT-comp-def GTT-trancl-def* Δ-*trancl-inv*)
qed

lemma *GTT-trancl-acomplete*:
gtrancl-rel UNIV (*agtt-lang G*) ⊆ *agtt-lang* (*GTT-trancl G*)
unfolding *gtrancl-rel-def*
using *agtt-lang-base*[*of G*] *gmctxt-cl-mono-rel*[*OF agtt-lang-base*[*of G*], *of UNIV*]
using *agtt-lang-trans*[*of G*]
unfolding *gcomp-rel-def*
by (*intro kleene-trancl-induct*) *blast*+

lemma *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*:

```
  (gtt-lang G)* = (gtt-lang G)+
  by (auto simp: rtrancl-trancl-reflcl simp del: reflcl-trancl dest: tranclD tranclD2
intro: gmctxt-cl-refl)

lemma GTT-trancl-complete:
  (gtt-lang G)+ ⊆ gtt-lang (GTT-trancl G)
 using GTT-trancl-base subset-trans[OF gtt-comp-lang-complete GTT-trancl-trans]
  by (metis trancl-id trancl-mono-set trans-O-iff)

lemma trancl-gtt-lang-arg-closed:
  assumes length ss = length ts ∀ i < length ts. (ss ! i, ts ! i) ∈ (gtt-lang 𝒢)+
  shows (GFun f ss, GFun f ts) ∈ (gtt-lang 𝒢)+ (is ?e ∈ -)
proof −
  have all-ctxt-closed UNIV ((gtt-lang 𝒢)+) by (intro all-ctxt-closed-trancl) auto
  from all-ctxt-closedD[OF this - assms] show ?thesis
    by auto
qed

lemma Δ-trancl-sound:
  assumes (p, q) |∈| Δ-trancl A B
  obtains s t where (s, t) ∈ (gtt-lang (B, A))+ p |∈| gta-der A s q |∈| gta-der B t
  using assms
proof (induct arbitrary: thesis rule: Δ-trancl-induct)
  case (Δ-cong f ps p qs q)
  have ∃ si ti. (si, ti) ∈ (gtt-lang (B, A))+ ∧ ps ! i |∈| gta-der A (si) ∧
      qs ! i |∈| gta-der B (ti) if i < length qs for i
    using Δ-cong(5)[OF that] by metis
  then obtain ss ts where
    ⋀i. i < length qs ⟹ (ss i, ts i) ∈ (gtt-lang (B, A))+ ∧ ps ! i |∈| gta-der A (ss
i) ∧ qs ! i |∈| gta-der B (ts i) by metis
  then show ?case using Δ-cong(1−5)
     by (intro Δ-cong(6)[of GFun f (map ss [0..<length ps]) GFun f (map ts
[0..<length qs])])
       (auto simp: gta-der-def intro!: trancl-gtt-lang-arg-closed)
next
  case (Δ-eps1 p p′ q) then show ?case
    by (metis gta-der-def ta-der-eps)
next
  case (Δ-eps2 q q′ p) then show ?case
    by (metis gta-der-def ta-der-eps)
next
  case (Δ-trans p q r)
 obtain s1 t1 where (s1, t1) ∈ (gtt-lang (B, A))+ p |∈| gta-der A s1 q |∈| gta-der
B t1
    using Δ-trans(2) .note 1 = this
 obtain s2 t2 where (s2, t2) ∈ (gtt-lang (B, A))+ q |∈| gta-der A s2 r |∈| gta-der
B t2
    using Δ-trans(4) . note 2 = this
  have (t1, s2) ∈ gtt-lang (B, A) using 1(1,3) 2(1,2)
```

by (*auto simp*: *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*[*symmetric*] *gtt-lang-join*)
then have (*s1*, *t2*) ∈ (*gtt-lang* (*B*, *A*))$^+$ **using** *1*(*1*) *2*(*1*)
by (*meson trancl.trancl-into-trancl trancl-trans*)
then show *?case* **using** *1*(*2*) *2*(*3*) **by** (*auto intro*: Δ-*trans*(*5*)[*of s1 t2*])
qed

lemma *GTT-trancl-sound-aux*:
assumes *p* |∈| *gta-der* (*TA* (*rules A*) (*eps A* |∪| (Δ-*trancl B A*))) *s*
shows ∃ *t*. (*s*, *t*) ∈ (*gtt-lang* (*A*, *B*))$^+$ ∧ *p* |∈| *gta-der A t*
using *assms*
proof (*induct s arbitrary*: *p*)
case (*GFun f ss*)
let *?eps* = *eps A* |∪| Δ-*trancl B A*
obtain *qs q* **where** *q*: *TA-rule f qs q* |∈| *rules A q* = *p* ∨ (*q*, *p*) |∈| *?eps*|$^+$| *length qs* = *length ss*
⋀*i*. *i* < *length ss* ⟹ *qs* ! *i* |∈| *gta-der* (*TA* (*rules A*) *?eps*) (*ss* ! *i*)
using *GFun*(*2*) **by** (*auto simp*: *gta-der-def*)
have ⋀*i*. *i* < *length ss* ⟹ ∃ *ti*. (*ss* ! *i*, *ti*) ∈ (*gtt-lang* (*A*, *B*))$^+$ ∧ *qs* ! *i* |∈| *gta-der A* (*ti*)
using *GFun*(*1*)[*OF nth-mem q*(*4*)] **unfolding** *gta-der-def* **by** *fastforce*
then obtain *ts* **where** *ts*: ⋀*i*. *i* < *length ss* ⟹ (*ss* ! *i*, *ts i*) ∈ (*gtt-lang* (*A*, *B*))$^+$ ∧ *qs* ! *i* |∈| *gta-der A* (*ts i*)
by *metis*
then have *q'*: *q* |∈| *gta-der A* (*GFun f* (*map ts* [*0*..<*length ss*]))
(*GFun f ss*, *GFun f* (*map ts* [*0*..<*length ss*])) ∈ (*gtt-lang* (*A*, *B*))$^+$ **using** *q*(*1*, *3*)
by (*auto simp*: *gta-der-def intro*!: *exI*[*of* - *qs*] *exI*[*of* - *q*] *trancl-gtt-lang-arg-closed*)
{fix *p q u* **assume** *ass*: (*p*, *q*) |∈| Δ-*trancl B A* (*GFun f ss*, *u*) ∈ (*gtt-lang* (*A*, *B*))$^+$ ∧ *p* |∈| *gta-der A u*
from Δ-*trancl-sound*[*OF this*(*1*)] **obtain** *s t*
where (*s*, *t*) ∈ (*gtt-lang* (*A*, *B*))$^+$ *p* |∈| *gta-der B s q* |∈| *gta-der A t* **. note** *st* = *this*
have (*u*, *s*) ∈ *gtt-lang* (*A*, *B*) **using** *st conjunct2*[*OF ass*(*2*)]
by (*auto simp*: *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*[*symmetric*] *gtt-lang-join*)
then have (*GFun f ss*, *t*) ∈ (*gtt-lang* (*A*, *B*))$^+$
using *ass st*(*1*) **by** (*meson trancl-into-trancl2 trancl-trans*)
then have ∃ *s t*. (*GFun f ss*, *t*) ∈ (*gtt-lang* (*A*, *B*))$^+$ ∧ *q* |∈| *gta-der A t* **using** *st* **by** *blast***}**
note *trancl-step* = *this*
show *?case*
proof (*cases q* = *p*)
case *True*
then show *?thesis* **using** *ts q*(*1*, *3*)
by (*auto simp*: *gta-der-def intro*!: *exI*[*of* -*GFun f* (*map ts* [*0*..< *length ss*])] *trancl-gtt-lang-arg-closed*) *blast*
next
case *False*
then have (*q*, *p*) |∈| *?eps*|$^+$| **using** *q*(*2*) **by** *simp*
then show *?thesis* **using** *q*(*1*) *q'*

```
    proof (induct rule: ftrancl-induct)
      case (Base q p) from Base(1) show ?case
      proof
        assume (q, p) |∈| eps A then show ?thesis using Base(2) ts q(3)
          by (auto simp: gta-der-def intro!: exI[of -GFun f (map ts [0..< length ss])]
                         trancl-gtt-lang-arg-closed exI[of - qs] exI[of - q])
      next
        assume (q, p) |∈| (Δ-trancl B A)
        then have (q, p) |∈| Δ-trancl B A by simp
        from trancl-step[OF this] show ?thesis using Base(3, 4)
          by auto
      qed
    next
      case (Step p q r)
      from Step(2, 4−) obtain s' where s': (GFun f ss, s') ∈ (gtt-lang (A, B))+
∧ q |∈| gta-der A s' by auto
      show ?case using Step(3)
      proof
        assume (q, r) |∈| eps A then show ?thesis using s'
          by (auto simp: gta-der-def ta-der-eps intro!: exI[of - s'])
      next
        assume (q, r) |∈| Δ-trancl B A
        then have (q, r) |∈| Δ-trancl B A by simp
        from trancl-step[OF this] show ?thesis using s' by auto
      qed
    qed
  qed
qed

lemma GTT-trancl-asound:
  agtt-lang (GTT-trancl G) ⊆ gtrancl-rel UNIV (agtt-lang G)
proof (intro subrelI, goal-cases LR)
  case (LR s t)
  then obtain s' q t' where ∗: (s, s') ∈ (gtt-lang G)+
    q |∈| gta-der (fst G) s' q |∈| gta-der (snd G) t' (t', t) ∈ (gtt-lang G)+
    by (auto simp: agtt-lang-def GTT-trancl-def trancl-converse Δ-trancl-inv
      simp flip: gtt-lang-swap[of fst G snd G, unfolded prod.collapse agtt-lang-def,
simplified]
      dest!: GTT-trancl-sound-aux)
  then have (s', t') ∈ agtt-lang G using ∗(2,3)
    by (auto simp: agtt-lang-def)
  then show ?case using ∗(1,4) unfolding gtrancl-rel-def
    by auto
qed

lemma GTT-trancl-sound:
  gtt-lang (GTT-trancl G) ⊆ (gtt-lang G)+
proof −
  note [dest] = GTT-trancl-sound-aux
```

have *gtt-accept* (*GTT-trancl G*) *s t* ⟹ (*s*, *t*) ∈ (*gtt-lang G*)$^+$ **for** *s t* **unfolding** *gtt-accept-def*
proof (*induct rule*: *gmctxt-cl.induct*)
case (*base s t*)
from *base* **obtain** *q* **where** *join*: *q* |∈| *gta-der* (*fst* (*GTT-trancl G*)) *s q* |∈| *gta-der* (*snd* (*GTT-trancl G*)) *t*
by (*auto simp*: *agtt-lang-def*)
obtain *s'* **where** (*s*, *s'*) ∈ (*gtt-lang G*)$^+$ *q* |∈| *gta-der* (*fst G*) *s'* **using** *base join*
by (*auto simp*: *GTT-trancl-def* Δ*-trancl-inv agtt-lang-def*)
moreover obtain *t'* **where** (*t'*, *t*) ∈ (*gtt-lang G*)$^+$ *q* |∈| *gta-der* (*snd G*) *t'* **using** *join*
by (*auto simp*: *GTT-trancl-def gtt-lang-swap*[*of fst G snd G*, *symmetric*] *trancl-converse* Δ*-trancl-inv*)
moreover have (*s'*, *t'*) ∈ *gtt-lang G* **using** *calculation*
by (*auto simp*: *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*[*symmetric*] *gtt-lang-join*)
ultimately show (*s*, *t*) ∈ (*gtt-lang G*)$^+$ **by** (*meson trancl.trancl-into-trancl trancl-trans*)
qed (*auto intro*!: *trancl-gtt-lang-arg-closed*)
then show *?thesis* **by** (*auto simp*: *gtt-accept-def*)
qed

lemma *GTT-trancl-alang*:
agtt-lang (*GTT-trancl G*) = *gtrancl-rel UNIV* (*agtt-lang G*)
using *GTT-trancl-asound GTT-trancl-acomplete* **by** *blast*

lemma *GTT-trancl-lang*:
gtt-lang (*GTT-trancl G*) = (*gtt-lang G*)$^+$
using *GTT-trancl-sound GTT-trancl-complete* **by** *blast*

end
theory *Pair-Automaton*
imports *Tree-Automata-Complement GTT-Compose*
begin

4.9 Pair automaton and anchored GTTs

definition *pair-at-lang* :: ('*q*, '*f*) *gtt* ⇒ ('*q* × '*q*) *fset* ⇒ '*f gterm rel* **where**
pair-at-lang 𝒢 *Q* = {(*s*, *t*) | *s t p q*. *q* |∈| *gta-der* (*fst* 𝒢) *s* ∧ *p* |∈| *gta-der* (*snd* 𝒢) *t* ∧ (*q*, *p*) |∈| *Q*}

lemma *pair-at-lang-restr-states*:
pair-at-lang 𝒢 *Q* = *pair-at-lang* 𝒢 (*Q* |∩| (𝒬 (*fst* 𝒢) |×| 𝒬 (*snd* 𝒢)))
by (*auto simp*: *pair-at-lang-def gta-der-def*) (*meson gterm-ta-der-states*)

lemma *pair-at-langE*:
assumes (*s*, *t*) ∈ *pair-at-lang* 𝒢 *Q*
obtains *q p* **where** (*q*, *p*) |∈| *Q* **and** *q* |∈| *gta-der* (*fst* 𝒢) *s* **and** *p* |∈| *gta-der* (*snd* 𝒢) *t*
using *assms* **by** (*auto simp*: *pair-at-lang-def*)

lemma *pair-at-langI*:
assumes $q \mid\in\mid$ *gta-der* (*fst* $\mathcal{G}$) s $p \mid\in\mid$ *gta-der* (*snd* $\mathcal{G}$) t $(q, p) \mid\in\mid Q$
shows $(s, t) \in$ *pair-at-lang* $\mathcal{G}$ Q
using *assms* **by** (*auto simp*: *pair-at-lang-def*)

lemma *pair-at-lang-fun-states*:
assumes *finj-on f* ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) **and** *finj-on g* ($\mathcal{Q}$ (*snd* $\mathcal{G}$))
and $Q \mid\subseteq\mid \mathcal{Q}$ (*fst* $\mathcal{G}$) $\mid\times\mid \mathcal{Q}$ (*snd* $\mathcal{G}$)
shows *pair-at-lang* $\mathcal{G}$ Q = *pair-at-lang* (*map-prod* (*fmap-states-ta f*) (*fmap-states-ta g*) $\mathcal{G}$) (*map-prod f g* $\mid`\mid$ Q)
(**is** *?LS* = *?RS*)
proof
{fix s t **assume** $(s, t) \in$ *?LS*
then have $(s, t) \in$ *?RS* **using** *ta-der-fmap-states-ta-mono*[*of f fst* $\mathcal{G}$ s]
using *ta-der-fmap-states-ta-mono*[*of g snd* $\mathcal{G}$ t]
by (*force simp*: *gta-der-def map-prod-def image-iff elim*!: *pair-at-langE split*: *prod.split intro*!: *pair-at-langI*)**}**
then show *?LS* $\subseteq$ *?RS* **by** *auto*
next
{fix s t **assume** $(s, t) \in$ *?RS*
then obtain p q **where** *rs*: $p \mid\in\mid$ *ta-der* (*fst* $\mathcal{G}$) (*term-of-gterm s*) *f p* $\mid\in\mid$ *ta-der* (*fmap-states-ta f* (*fst* $\mathcal{G}$)) (*term-of-gterm s*) **and**
ts: $q \mid\in\mid$ *ta-der* (*snd* $\mathcal{G}$) (*term-of-gterm t*) *g q* $\mid\in\mid$ *ta-der* (*fmap-states-ta g* (*snd* $\mathcal{G}$)) (*term-of-gterm t*) **and**
st: (*f p*, *g q*) $\mid\in\mid$ (*map-prod f g* $\mid`\mid$ Q) **using** *assms ta-der-fmap-states-inv*[*of f fst* $\mathcal{G}$ - *s*]
using *ta-der-fmap-states-inv*[*of g snd* $\mathcal{G}$ - *t*]
by (*auto simp*: *gta-der-def adapt-vars-term-of-gterm elim*!: *pair-at-langE*)
(*metis* (*no-types*, *opaque-lifting*) *f-the-finv-into-f fimage.rep-eq fmap-prod-fimageI fmap-states gterm-ta-der-states*)
then have $p \mid\in\mid \mathcal{Q}$ (*fst* $\mathcal{G}$) $q \mid\in\mid \mathcal{Q}$ (*snd* $\mathcal{G}$) **by** *auto*
then have $(p, q) \mid\in\mid Q$ **using** *assms st* **unfolding** *fimage-iff fBex-def*
by (*auto dest*!: *fsubsetD simp*: *finj-on-eq-iff*)
then have $(s, t) \in$ *?LS* **using** *st rs*(*1*) *ts*(*1*) **by** (*auto simp*: *gta-der-def intro*!: *pair-at-langI*)**}**
then show *?RS* $\subseteq$ *?LS* **by** *auto*
qed

lemma *converse-pair-at-lang*:
(*pair-at-lang* $\mathcal{G}$ Q)$^{-1}$ = *pair-at-lang* (*prod.swap* $\mathcal{G}$) ($Q|^{-1}|$)
by (*auto simp*: *pair-at-lang-def*)

lemma *pair-at-agtt*:
agtt-lang $\mathcal{G}$ = *pair-at-lang* $\mathcal{G}$ (*fId-on* (*gtt-interface* $\mathcal{G}$))
by (*auto simp*: *agtt-lang-def gtt-interface-def pair-at-lang-def gtt-states-def gta-der-def fId-on-iff*)

definition Δ-*eps-pair* **where**

Δ-*eps-pair* $\mathcal{G}_1$ Q_1 $\mathcal{G}_2$ Q_2 $\equiv$ Q_1 $|O|$ Δ_ε (*snd* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$) $|O|$ Q_2

lemma *pair-comp-sound1*:
assumes $(s, t) \in$ *pair-at-lang* $\mathcal{G}_1$ Q_1
and $(t, u) \in$ *pair-at-lang* $\mathcal{G}_2$ Q_2
shows $(s, u) \in$ *pair-at-lang* (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) (Δ-*eps-pair* $\mathcal{G}_1$ Q_1 $\mathcal{G}_2$ Q_2)
proof −
from *pair-at-langE assms* **obtain** p q q' r **where**
wit: $(p, q) |\in| Q_1\ p |\in|$ *gta-der* (*fst* $\mathcal{G}_1$) $s\ q |\in|$ *gta-der* (*snd* $\mathcal{G}_1$) t
$(q', r) |\in| Q_2\ q' |\in|$ *gta-der* (*fst* $\mathcal{G}_2$) $t\ r |\in|$ *gta-der* (*snd* $\mathcal{G}_2$) u
by *metis*
from *wit*(*3*, *5*) **have** $(q, q') |\in| \Delta_\varepsilon$ (*snd* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$)
by (*auto simp*: Δ_ε-*def'* *gta-der-def intro*!: *exI*[*of* - *term-of-gterm t*])
then have $(p, r) |\in|$ Δ-*eps-pair* $\mathcal{G}_1$ Q_1 $\mathcal{G}_2$ Q_2 **using** *wit*(*1*, *4*)
by (*auto simp*: Δ-*eps-pair-def*)
then show *?thesis* **using** *wit*(*2*, *6*) **unfolding** *pair-at-lang-def*
by *auto*
qed

lemma *pair-comp-sound2*:
assumes $(s, u) \in$ *pair-at-lang* (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) (Δ-*eps-pair* $\mathcal{G}_1$ Q_1 $\mathcal{G}_2$ Q_2)
shows $\exists\ t.\ (s, t) \in$ *pair-at-lang* $\mathcal{G}_1$ $Q_1 \wedge (t, u) \in$ *pair-at-lang* $\mathcal{G}_2$ Q_2
using *assms* **unfolding** *pair-at-lang-def* Δ-*eps-pair-def*
by (*auto simp*: Δ_ε-*def'* *gta-der-def*) (*metis gterm-of-term-inv*)

lemma *pair-comp-sound*:
pair-at-lang $\mathcal{G}_1$ Q_1 *O pair-at-lang* $\mathcal{G}_2$ Q_2 = *pair-at-lang* (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) (Δ-*eps-pair* $\mathcal{G}_1$ Q_1 $\mathcal{G}_2$ Q_2)
by (*auto simp*: *pair-comp-sound1 pair-comp-sound2 relcomp.simps*)

inductive-set Δ-*Atrans-set* :: ('q × 'q) *fset* ⇒ ('q, 'f) *ta* ⇒ ('q, 'f) *ta* ⇒ ('q × 'q) *set* **for** Q $\mathcal{A}$ $\mathcal{B}$ **where**
base [*simp*]: $(p, q) |\in| Q \Longrightarrow (p, q) \in \Delta$-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B}$
| *step* [*intro*]: $(p, q) \in \Delta$-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B} \Longrightarrow (q, r) |\in| \Delta_\varepsilon$ $\mathcal{B}$ $\mathcal{A} \Longrightarrow$
$(r, v) \in \Delta$-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B} \Longrightarrow (p, v) \in \Delta$-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B}$

lemma Δ-*Atrans-set-states*:
$(p, q) \in \Delta$-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B} \Longrightarrow (p, q) \in$ *fset* ((*fst* |‘| Q |∪| $\mathcal{Q}$ $\mathcal{A}$) |×| (*snd* |‘| Q |∪| $\mathcal{Q}$ $\mathcal{B}$))
by (*induct rule*: Δ-*Atrans-set.induct*) (*auto simp*: *image-iff intro*!: *bexI*)

lemma *finite-*Δ-*Atrans-set*: *finite* (Δ-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B}$)
proof −
have Δ-*Atrans-set* Q $\mathcal{A}$ $\mathcal{B} \subseteq$ *fset* ((*fst* |‘| Q |∪| $\mathcal{Q}$ $\mathcal{A}$) |×| (*snd* |‘| Q |∪| $\mathcal{Q}$ $\mathcal{B}$))
using Δ-*Atrans-set-states*
by (*metis subrelI*)
from *finite-subset*[*OF this*] **show** *?thesis* **by** *simp*
qed

context
includes *fset.lifting*
begin
lift-definition Δ-*Atrans* :: $('q \times 'q)$ *fset* ⇒ $('q, 'f)$ *ta* ⇒ $('q, 'f)$ *ta* ⇒ $('q \times 'q)$ *fset* **is** Δ-*Atrans-set*
by (*simp add*: *finite-*Δ-*Atrans-set*)

lemmas Δ-*Atrans-base* [*simp*] = Δ-*Atrans-set.base* [*Transfer.transferred*]
lemmas Δ-*Atrans-step* [*intro*] = Δ-*Atrans-set.step* [*Transfer.transferred*]
lemmas Δ-*Atrans-cases* = Δ-*Atrans-set.cases*[*Transfer.transferred*]
lemmas Δ-*Atrans-induct* [*consumes 1*, *case-names base step*] = Δ-*Atrans-set.induct*[*Transfer.transferred*]
end

abbreviation Δ-*Atrans-gtt* $\mathcal{G}$ Q ≡ Δ-*Atrans* Q (*fst* $\mathcal{G}$) (*snd* $\mathcal{G}$)

lemma *pair-trancl-sound1*:
assumes $(s, t) \in$ (*pair-at-lang* $\mathcal{G}$ Q)$^+$
shows ∃ q p. p |∈| *gta-der* (*fst* $\mathcal{G}$) s ∧ q |∈| *gta-der* (*snd* $\mathcal{G}$) t ∧ (p, q) |∈| Δ-*Atrans-gtt* $\mathcal{G}$ Q
using *assms*
proof (*induct*)
case (*step t v*)
obtain p q r r' **where** *reach-t*: r |∈| *gta-der* (*fst* $\mathcal{G}$) t q |∈| *gta-der* (*snd* $\mathcal{G}$) t **and**
reach: p |∈| *gta-der* (*fst* $\mathcal{G}$) s r' |∈| *gta-der* (*snd* $\mathcal{G}$) v **and**
st: (p, q) |∈| Δ-*Atrans-gtt* $\mathcal{G}$ Q (r, r') |∈| Q **using** *step*(*2*, *3*)
by (*auto simp*: *pair-at-lang-def*)
from *reach-t* **have** (q, r) |∈| Δ_ε (*snd* $\mathcal{G}$) (*fst* $\mathcal{G}$)
by (*auto simp*: Δ_ε-*def'* *gta-der-def intro*: *ground-term-of-gterm*)
then have (p, r') |∈| Δ-*Atrans-gtt* $\mathcal{G}$ Q **using** *st* **by** *auto*
then show *?case* **using** *reach reach-t*
by (*auto simp*: *pair-at-lang-def gta-der-def* Δ_ε-*def'* *intro*: *ground-term-of-gterm*)
qed (*auto simp*: *pair-at-lang-def intro*: Δ-*Atrans-base*)

lemma *pair-trancl-sound2*:
assumes (p, q) |∈| Δ-*Atrans-gtt* $\mathcal{G}$ Q
and p |∈| *gta-der* (*fst* $\mathcal{G}$) s q |∈| *gta-der* (*snd* $\mathcal{G}$) t
shows $(s, t) \in$ (*pair-at-lang* $\mathcal{G}$ Q)$^+$ **using** *assms*
proof (*induct arbitrary*: s t *rule*:Δ-*Atrans-induct*)
case (*step p q r v*)
from *step*(*2*)[*OF step*(*6*)] *step*(*5*)[*OF - step*(*7*)] *step*(*3*)
show *?case* **by** (*auto simp*: *gta-der-def* Δ_ε-*def'* *intro*!: *ground-term-of-gterm*)
(*metis gterm-of-term-inv trancl-trans*)
qed (*auto simp*: *pair-at-lang-def*)

lemma *pair-trancl-sound*:
(*pair-at-lang* $\mathcal{G}$ Q)$^+$ = *pair-at-lang* $\mathcal{G}$ (Δ-*Atrans-gtt* $\mathcal{G}$ Q)
by (*auto simp*: *pair-trancl-sound2 dest*: *pair-trancl-sound1 elim*: *pair-at-langE intro*: *pair-at-langI*)

abbreviation *fst-pair-cl* $\mathcal{A}$ $Q \equiv$ *TA* (*rules* $\mathcal{A}$) (*eps* $\mathcal{A}$ |∪| (*fId-on* ($\mathcal{Q}$ $\mathcal{A}$) |*O*| Q))
definition *pair-at-to-agtt* :: ('*q*, '*f*) *gtt* ⇒ ('*q* × '*q*) *fset* ⇒ ('*q*, '*f*) *gtt* **where**
pair-at-to-agtt $\mathcal{G}$ Q = (*fst-pair-cl* (*fst* $\mathcal{G}$) Q , *TA* (*rules* (*snd* $\mathcal{G}$)) (*eps* (*snd* $\mathcal{G}$)))

lemma *fst-pair-cl-eps*:
assumes (p, q) |∈| (*eps* (*fst-pair-cl* $\mathcal{A}$ Q))|$^+$|
and $\mathcal{Q}$ $\mathcal{A}$ |∩| *snd* |‘| Q = {||}
shows (p, q) |∈| (*eps* $\mathcal{A}$)|$^+$| ∨ (∃ r. (p = r ∨ (p, r) |∈| (*eps* $\mathcal{A}$)|$^+$|) ∧ (r, q) |∈| Q) **using** *assms*
proof (*induct rule*: *ftrancl-induct*)
case (*Step p q r*)
then have *y*: q |∈| $\mathcal{Q}$ $\mathcal{A}$ **by** (*auto simp add*: *eps-trancl-statesD eps-statesD*)
have [*simp*]: (p, q) |∈| Q ⟹ q |∈| *snd* |‘| Q **for** p q **by** (*auto simp*: *fimage-iff*) *force*
then show *?case* **using** *Step y*
by *auto* (*simp add*: *ftrancl-into-trancl*)
qed *auto*

lemma *fst-pair-cl-res-aux*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| *snd* |‘| Q = {||}
and q |∈| *ta-der* (*fst-pair-cl* $\mathcal{A}$ Q) (*term-of-gterm t*)
shows ∃ p. p |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ∧ (q |∉| $\mathcal{Q}$ $\mathcal{A}$ ⟶ (p, q) |∈| Q) ∧ (q |∈| $\mathcal{Q}$ $\mathcal{A}$ ⟶ p = q) **using** *assms*
proof (*induct t arbitrary*: q)
case (*GFun f ts*)
then obtain *qs* q' **where** *rule*: *TA-rule f qs* q' |∈| *rules* $\mathcal{A}$ *length qs* = *length ts* **and**
eps: q' = q ∨ (q', q) |∈| (*eps* (*fst-pair-cl* $\mathcal{A}$ Q))|$^+$| **and**
reach: ∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der* (*fst-pair-cl* $\mathcal{A}$ Q) (*term-of-gterm* (*ts* ! *i*))
by *auto*
{fix *i* **assume** *ass*: *i* < *length ts* **then have** *st*: *qs* ! *i* |∈| $\mathcal{Q}$ $\mathcal{A}$ **using** *rule*
by (*auto simp*: *rule-statesD*)
then have *qs* ! *i* |∉| *snd* |‘| Q **using** *GFun*(*2*) **by** *auto*
then have *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ts* ! *i*)) **using** *reach st ass*
using *fst-pair-cl-eps*[*OF* - *GFun*(*2*)] *GFun*(*1*)[*OF nth-mem*[*OF ass*] *GFun*(*2*), *of qs* ! *i*]
by *blast*} **note** *IH* = *this*
show *?case*
proof (*cases* q' = q)
case *True*
then show *?thesis* **using** *rule reach IH*
by (*auto dest*: *rule-statesD intro*!: *exI*[*of* - q'] *exI*[*of* - *qs*])
next
case *False* **note** *nt-eq* = *this*
then have *eps*: (q', q) |∈| (*eps* (*fst-pair-cl* $\mathcal{A}$ Q))|$^+$| **using** *eps* **by** *simp*
from *fst-pair-cl-eps*[*OF this assms*(*1*)] **show** *?thesis*
using *False rule IH*

```
    proof (cases q |∉| 𝒬 𝒜)
      case True
      from fst-pair-cl-eps[OF eps assms(1)] obtain r where
        q' = r ∨ (q', r) |∈| (eps 𝒜)|^+| (r, q) |∈| Q using True
        by (auto simp: eps-trancl-statesD)
      then show ?thesis using nt-eq rule IH True
        by (auto simp: fimage-iff eps-trancl-statesD)
    next
      case False
      from fst-pair-cl-eps[OF eps assms(1)] False assms(1)
      have (q', q) |∈| (eps 𝒜)|^+|
        by (auto simp: fimage-iff) (metis fempty-iff fimage-eqI finterI snd-conv)+
      then show ?thesis using IH rule
        by (intro exI[of - q]) (auto simp: eps-trancl-statesD)
    qed
  qed
qed

lemma restr-distjoing:
  assumes Q |⊆| 𝒬 𝒜 |×| 𝒬 ℬ
    and 𝒬 𝒜 |∩| 𝒬 ℬ = {||}
  shows 𝒬 𝒜 |∩| snd |`| Q = {||}
  using assms by auto

lemma pair-at-agtt-conv:
  assumes Q |⊆| 𝒬 (fst 𝒢) |×| 𝒬 (snd 𝒢) and 𝒬 (fst 𝒢) |∩| 𝒬 (snd 𝒢) = {||}
  shows pair-at-lang 𝒢 Q = agtt-lang (pair-at-to-agtt 𝒢 Q) (is ?LS = ?RS)
proof
  let ?TA = fst-pair-cl (fst 𝒢) Q
  {fix s t assume ls: (s, t) ∈ ?LS
    then obtain q p where w: (q, p) |∈| Q q |∈| gta-der (fst 𝒢) s p |∈| gta-der
(snd 𝒢) t
      by (auto elim: pair-at-langE)
    from w(2) have q |∈| gta-der ?TA s q |∈| 𝒬 (fst 𝒢)
      using ta-der-mono'[of fst 𝒢 ?TA term-of-gterm s]
      by (auto simp add: fin-mono ta-subset-def gta-der-def in-mono)
    then have (s, t) ∈ ?RS using w(1, 3)
      by (auto simp: pair-at-to-agtt-def agtt-lang-def gta-der-def ta-der-eps intro!:
exI[of - p])
        (metis fId-onI frelcompI funionI2 ta.sel(2) ta-der-eps)}
  then show ?LS ⊆ ?RS by auto
next
  {fix s t assume ls: (s, t) ∈ ?RS
    then obtain q where w: q |∈| ta-der (fst-pair-cl (fst 𝒢) Q) (term-of-gterm s)
      q |∈| ta-der (snd 𝒢) (term-of-gterm t)
      by (auto simp: agtt-lang-def pair-at-to-agtt-def gta-der-def)
    from w(2) have q |∈| 𝒬 (snd 𝒢) q |∉| 𝒬 (fst 𝒢) using assms(2)
      by auto
    from fst-pair-cl-res-aux[OF restr-distjoing[OF assms] w(1)] this w(2)
```

have $(s,\ t) \in$ *?LS* **by** (*auto simp*: *agtt-lang-def pair-at-to-agtt-def gta-der-def intro*: *pair-at-langI*)**}**
then show *?RS* $\subseteq$ *?LS* **by** *auto*
qed

definition *pair-at-to-agtt′* **where**
pair-at-to-agtt′ $\mathcal{G}\ Q$ = (*let* $\mathcal{A}$ = *fmap-states-ta Inl* (*fst* $\mathcal{G}$) *in*
let $\mathcal{B}$ = *fmap-states-ta Inr* (*snd* $\mathcal{G}$) *in*
let Q' = Q |∩| ($\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)) *in*
pair-at-to-agtt ($\mathcal{A}$, $\mathcal{B}$) (*map-prod Inl Inr* |‘| Q'))

lemma *pair-at-agtt-cost*:
pair-at-lang $\mathcal{G}\ Q$ = *agtt-lang* (*pair-at-to-agtt′* $\mathcal{G}\ Q$)
proof −
let *?G* = (*fmap-states-ta CInl* (*fst* $\mathcal{G}$), *fmap-states-ta CInr* (*snd* $\mathcal{G}$))
let *?Q* = (Q |∩| ($\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)))
let *?Q′* = *map-prod CInl CInr* |‘| *?Q*
have ∗: *pair-at-lang* $\mathcal{G}\ Q$ = *pair-at-lang* $\mathcal{G}$ *?Q*
using *pair-at-lang-restr-states* **by** *blast*
have *pair-at-lang* $\mathcal{G}$ *?Q* = *pair-at-lang* (*map-prod* (*fmap-states-ta CInl*) (*fmap-states-ta CInr*) $\mathcal{G}$) (*map-prod CInl CInr* |‘| *?Q*)
by (*intro pair-at-lang-fun-states*[**where** *?*$\mathcal{G}$ = $\mathcal{G}$ **and** *?Q* = *?Q* **and** *?f* = *CInl* **and** *?g* = *CInr*])
(*auto simp*: *finj-CInl-CInr*)
then have ∗∗:*pair-at-lang* $\mathcal{G}$ *?Q* = *pair-at-lang ?G ?Q′* **by** (*simp add*: *map-prod-simp′*)
have *pair-at-lang ?G ?Q′* = *agtt-lang* (*pair-at-to-agtt ?G ?Q′*)
by (*intro pair-at-agtt-conv*[**where** *?*$\mathcal{G}$ = *?G*]) *auto*
then show *?thesis* **unfolding** ∗ ∗∗ *pair-at-to-agtt′-def Let-def*
by *simp*
qed

lemma Δ-*Atrans-states-stable*:
assumes Q |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
shows Δ-*Atrans-gtt* $\mathcal{G}\ Q$ |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
proof
fix s **assume** *ass*: s |∈| Δ-*Atrans-gtt* $\mathcal{G}\ Q$
then obtain $t\ u$ **where** *s*: $s = (t,\ u)$ **by** (*cases s*) *blast*
show s |∈| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$) **using** *ass assms* **unfolding** *s*
by (*induct rule*: Δ-*Atrans-induct*) *auto*
qed

lemma Δ-*Atrans-map-prod*:
assumes *finj-on f* ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) **and** *finj-on g* ($\mathcal{Q}$ (*snd* $\mathcal{G}$))
and Q |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
shows *map-prod f g* |‘| (Δ-*Atrans-gtt* $\mathcal{G}\ Q$) = Δ-*Atrans-gtt* (*map-prod* (*fmap-states-ta f*) (*fmap-states-ta g*) $\mathcal{G}$) (*map-prod f g* |‘| Q)
(**is** *?LS* = *?RS*)
proof −
{fix $p\ q$ **assume** $(p,\ q)$ |∈| Δ-*Atrans-gtt* $\mathcal{G}\ Q$

```
    then have (f p, g q) |∈| ?RS using assms
    proof (induct rule: Δ-Atrans-induct)
      case (step p q r v)
        from step(3, 6, 7) have (g q, f r) |∈| Δ_ε (fmap-states-ta g (snd 𝒢))
(fmap-states-ta f (fst 𝒢))
        by (auto simp: Δ_ε-def' intro!: ground-term-of-gterm)
        (metis ground-term-of-gterm ground-term-to-gtermD ta-der-to-fmap-states-der)
      then show ?case using step by auto
    qed (auto simp add: map-prod-imageI)}
  moreover
  {fix p q assume (p, q) |∈| ?RS
    then have (p, q) |∈| ?LS using assms
    proof (induct rule: Δ-Atrans-induct)
      case (step p q r v)
      let ?f = the-finv-into (𝒬 (fst 𝒢)) f let ?g = the-finv-into (𝒬 (snd 𝒢)) g
      have sub: Δ_ε (snd 𝒢) (fst 𝒢) |⊆| 𝒬 (snd 𝒢) |×| 𝒬 (fst 𝒢)
        using Δ_ε-statesD(1, 2) by fastforce
      have s-e: (?f p, ?g q) |∈| Δ-Atrans-gtt 𝒢 Q (?f r, ?g v) |∈| Δ-Atrans-gtt 𝒢 Q
        using step assms(1, 2) fsubsetD[OF Δ-Atrans-states-stable[OF assms(3)]]
        using finj-on-eq-iff[OF assms(1)] finj-on-eq-iff
        using the-finv-into-f-f[OF assms(1)] the-finv-into-f-f[OF assms(2)]
        by auto
      from step(3) have (?g q, ?f r) |∈| Δ_ε (snd 𝒢) (fst 𝒢)
        using step(6−) sub
          using ta-der-fmap-states-conv[OF assms(1)] ta-der-fmap-states-conv[OF
assms(2)]
        using the-finv-into-f-f[OF assms(1)] the-finv-into-f-f[OF assms(2)]
        by (auto simp: Δ_ε-fmember fimage-iff fBex-def)
        (metis ground-term-of-gterm ground-term-to-gtermD ta-der-fmap-states-inv)
      then have (q, r) |∈| map-prod g f |`| Δ_ε (snd 𝒢) (fst 𝒢) using step
        using the-finv-into-f-f[OF assms(1)] the-finv-into-f-f[OF assms(2)] sub
       by (smt (verit, ccfv-threshold) Δ_ε-statesD(1) Δ_ε-statesD(2) f-the-finv-into-f
fimage.rep-eq
          fmap-states fst-map-prod map-prod-imageI snd-map-prod)
      then show ?case using s-e assms(1, 2) s-e
        using fsubsetD[OF sub]
        using fsubsetD[OF Δ-Atrans-states-stable[OF assms(3)]]
        using Δ-Atrans-step[of ?f p ?g q Q fst 𝒢 snd 𝒢 ?f r ?g v]
        using the-finv-into-f-f[OF assms(1)] the-finv-into-f-f[OF assms(2)]
        using step.hyps(2) step.hyps(5) step.prems(3) by force
    qed auto}
  ultimately show ?thesis by auto
qed

— Section: Pair Automaton is closed under Determinization

definition Q-pow where
  Q-pow Q 𝒮₁ 𝒮₂ =
    {|(Wrapp X, Wrapp Y) | X Y p q. X |∈| fPow 𝒮₁ ∧ Y |∈| fPow 𝒮₂ ∧ p |∈| X
```

$\wedge$ q $|\in|$ Y $\wedge$ $(p,\ q)$ $|\in|$ $Q|\}$

lemma *Q-pow-fmember*:
$(X,\ Y)$ $|\in|$ *Q-pow* Q $\mathcal{S}_1$ $\mathcal{S}_2$ $\longleftrightarrow$ $(\exists\ p\ q.$ *ex* X $|\in|$ *fPow* $\mathcal{S}_1$ $\wedge$ *ex* Y $|\in|$ *fPow* $\mathcal{S}_2$ $\wedge$ p $|\in|$ *ex* X $\wedge$ q $|\in|$ *ex* Y $\wedge$ $(p,\ q)$ $|\in|$ $Q)$
proof −
let *?S* = $\{($*Wrapp* $X,$ *Wrapp* $Y)$ | X Y p $q.$ X $|\in|$ *fPow* $\mathcal{S}_1$ $\wedge$ Y $|\in|$ *fPow* $\mathcal{S}_2$ $\wedge$ p $|\in|$ X $\wedge$ q $|\in|$ Y $\wedge$ $(p,\ q)$ $|\in|$ $Q\}$
have *?S* $\subseteq$ *map-prod Wrapp Wrapp* ‘ *fset* (*fPow* $\mathcal{S}_1$ $|\times|$ *fPow* $\mathcal{S}_2$) **by** *auto*
from *finite-subset*[*OF this*] **show** *?thesis* **unfolding** *Q-pow-def*
apply *auto* **apply** *blast*
by (*meson FSet-Lex-Wrapper.exhaust-sel*)
qed

lemma *pair-automaton-det-lang-sound-complete*:
pair-at-lang $\mathcal{G}$ Q = *pair-at-lang* (*map-both ps-ta* $\mathcal{G}$) (*Q-pow* Q ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) ($\mathcal{Q}$ (*snd* $\mathcal{G}$))) (**is** *?LS* = *?RS*)
proof −
{fix s t **assume** $(s,\ t)$ ∈ *?LS*
then obtain p q **where**
res : p $|\in|$ *ta-der* (*fst* $\mathcal{G}$) (*term-of-gterm* s)
q $|\in|$ *ta-der* (*snd* $\mathcal{G}$) (*term-of-gterm* t) $(p,\ q)$ $|\in|$ Q
by (*auto simp*: *pair-at-lang-def gta-der-def*)
from *ps-rules-complete*[*OF this*(*1*)] *ps-rules-complete*[*OF this*(*2*)] *this*(*3*)
have $(s,\ t)$ ∈ *?RS* **using** *fPow-iff ps-ta-states′*
apply (*auto simp*: *pair-at-lang-def gta-der-def Q-pow-fmember*)
by (*smt* (*verit*, *best*) *dual-order.trans ground-ta-der-states ground-term-of-gterm ps-rules-sound*)**}**
moreover
{fix s t **assume** $(s,\ t)$ ∈ *?RS* **then have** $(s,\ t)$ ∈ *?LS*
using *ps-rules-sound*
by (*auto simp*: *pair-at-lang-def gta-der-def ps-ta-def Let-def Q-pow-fmember*) *blast***}**
ultimately show *?thesis* **by** *auto*
qed

lemma *pair-automaton-complement-sound-complete*:
assumes *partially-completely-defined-on* $\mathcal{A}$ $\mathcal{F}$ **and** *partially-completely-defined-on* $\mathcal{B}$ $\mathcal{F}$
and *ta-det* $\mathcal{A}$ **and** *ta-det* $\mathcal{B}$
shows *pair-at-lang* $(\mathcal{A},\ \mathcal{B})$ ($\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathcal{B}$ $|-|$ Q) = *gterms* (*fset* $\mathcal{F}$) $\times$ *gterms* (*fset* $\mathcal{F}$) − *pair-at-lang* $(\mathcal{A},\ \mathcal{B})$ Q
using *assms* **unfolding** *partially-completely-defined-on-def pair-at-lang-def*
apply (*auto simp*: *gta-der-def*)
apply (*metis ta-detE*)
apply *fastforce*
done

end

theory *AGTT*
imports *GTT GTT-Transitive-Closure Pair-Automaton*
begin

definition *AGTT-union* **where**
AGTT-union $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ (*ta-union* (*fst* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$),
ta-union (*snd* $\mathcal{G}_1$) (*snd* $\mathcal{G}_2$))

abbreviation *AGTT-union′* **where**
AGTT-union′ $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *AGTT-union* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *disj-gtt-states-disj-fst-ta-states*:
assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows $\mathcal{Q}$ (*fst* $\mathcal{G}_1$) |∩| $\mathcal{Q}$ (*fst* $\mathcal{G}_2$) = {||}
using *assms* **unfolding** *gtt-states-def* **by** *auto*

lemma *disj-gtt-states-disj-snd-ta-states*:
assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows $\mathcal{Q}$ (*snd* $\mathcal{G}_1$) |∩| $\mathcal{Q}$ (*snd* $\mathcal{G}_2$) = {||}
using *assms* **unfolding** *gtt-states-def* **by** *auto*

lemma *ta-der-not-contains-undefined-state*:
assumes *q* |∉| $\mathcal{Q}$ *T* **and** *ground t*
shows *q* |∉| *ta-der T t*
using *ground-ta-der-states*[*OF assms*(*2*)] *assms*(*1*)
by *blast*

lemma *AGTT-union-sound1*:
assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$) ⊆ *agtt-lang* $\mathcal{G}_1$ ∪ *agtt-lang* $\mathcal{G}_2$
proof −
let *?TA-A* = *ta-union* (*fst* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$)
let *?TA-B* = *ta-union* (*snd* $\mathcal{G}_1$) (*snd* $\mathcal{G}_2$)
{fix *s t* **assume** *ass*: (*s*, *t*) ∈ *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$)
then obtain *q* **where** *ls*: *q* |∈| *ta-der ?TA-A* (*term-of-gterm s*) **and**
rs: *q* |∈| *ta-der ?TA-B* (*term-of-gterm t*)
by (*auto simp add*: *AGTT-union-def agtt-lang-def gta-der-def*)
then have (*s*, *t*) ∈ *agtt-lang* $\mathcal{G}_1$ ∨ (*s*, *t*) ∈ *agtt-lang* $\mathcal{G}_2$
proof (*cases q* |∈| *gtt-states* $\mathcal{G}_1$)
case *True*
then have *q* |∉| *gtt-states* $\mathcal{G}_2$ **using** *dist-st*
by *blast*
then have *nt-fst-st*: *q* |∉| $\mathcal{Q}$ (*fst* $\mathcal{G}_2$) **and**
nt-snd-state: *q* |∉| $\mathcal{Q}$ (*snd* $\mathcal{G}_2$) **by** (*auto simp add*: *gtt-states-def*)
from *True* **show** *?thesis*
using *ls rs*
using *ta-der-not-contains-undefined-state*[*OF nt-fst-st*]

```
      using ta-der-not-contains-undefined-state[OF nt-snd-state]
      unfolding gtt-states-def agtt-lang-def gta-der-def
     using ta-union-der-disj-states[OF disj-gtt-states-disj-fst-ta-states[OF dist-st]]
    using ta-union-der-disj-states[OF disj-gtt-states-disj-snd-ta-states[OF dist-st]]
      using ground-term-of-gterm by blast
  next
    case False
    then have q |∉| gtt-states G₁ by (metis IntI dist-st emptyE)
    then have nt-fst-st: q |∉| Q (fst G₁) and
      nt-snd-state: q |∉| Q (snd G₁) by (auto simp add: gtt-states-def)
    from False show ?thesis
      using ls rs
      using ta-der-not-contains-undefined-state[OF nt-fst-st]
      using ta-der-not-contains-undefined-state[OF nt-snd-state]
      unfolding gtt-states-def agtt-lang-def gta-der-def
     using ta-union-der-disj-states[OF disj-gtt-states-disj-fst-ta-states[OF dist-st]]
    using ta-union-der-disj-states[OF disj-gtt-states-disj-snd-ta-states[OF dist-st]]
      using ground-term-of-gterm by blast
  qed}
 then show ?thesis by auto
qed

lemma AGTT-union-sound2:
 shows agtt-lang G₁ ⊆ agtt-lang (AGTT-union G₁ G₂)
   agtt-lang G₂ ⊆ agtt-lang (AGTT-union G₁ G₂)
 unfolding agtt-lang-def gta-der-def AGTT-union-def
 by auto (meson fin-mono ta-der-mono' ta-union-ta-subset)+

lemma AGTT-union-sound:
 assumes dist-st: gtt-states G₁ |∩| gtt-states G₂ = {||}
 shows agtt-lang (AGTT-union G₁ G₂) = agtt-lang G₁ ∪ agtt-lang G₂
 using AGTT-union-sound1[OF assms] AGTT-union-sound2 by blast

lemma AGTT-union'-sound:
 fixes G₁ :: ('q, 'f) gtt and G₂ :: ('q, 'f) gtt
 shows agtt-lang (AGTT-union' G₁ G₂) = agtt-lang G₁ ∪ agtt-lang G₂
proof −
 have map: agtt-lang (AGTT-union' G₁ G₂) =
   agtt-lang (fmap-states-gtt CInl G₁) ∪ agtt-lang (fmap-states-gtt CInr G₂)
   by (intro  AGTT-union-sound) (auto simp add: agtt-lang-fmap-states-gtt)
 then show ?thesis by (simp add: agtt-lang-fmap-states-gtt finj-CInl-CInr)
qed
```

4.10 Anchord gtt compositon

```
definition AGTT-comp :: ('q, 'f) gtt ⇒ ('q, 'f) gtt ⇒ ('q, 'f) gtt where
 AGTT-comp G₁ G₂ = (let (A, B) = (fst G₁, snd G₂) in
    (TA (rules A) (eps A |∪| (Δₑ (snd G₁) (fst G₂) |∩| (gtt-interface G₁ |×|
gtt-interface G₂)))),
```

TA (*rules* $\mathcal{B}$) (*eps* $\mathcal{B}$)))

abbreviation *AGTT-comp'* **where**
AGTT-comp' $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *AGTT-comp* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *AGTT-comp-sound*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*AGTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ *O agtt-lang* $\mathcal{G}_2$
proof −
let *?*Q_1 = *fId-on* (*gtt-interface* $\mathcal{G}_1$) **let** *?*Q_2 = *fId-on* (*gtt-interface* $\mathcal{G}_2$)
have *lan*: *agtt-lang* $\mathcal{G}_1$ = *pair-at-lang* $\mathcal{G}_1$ *?*Q_1 *agtt-lang* $\mathcal{G}_2$ = *pair-at-lang* $\mathcal{G}_2$ *?*Q_2
using *pair-at-agtt*[*of* $\mathcal{G}_1$] *pair-at-agtt*[*of* $\mathcal{G}_2$]
by *auto*
have *agtt-lang* $\mathcal{G}_1$ *O agtt-lang* $\mathcal{G}_2$ = *pair-at-lang* (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) (Δ-*eps-pair* $\mathcal{G}_1$ *?*Q_1 $\mathcal{G}_2$ *?*Q_2)
using *pair-comp-sound1 pair-comp-sound2*
by (*auto simp add*: *lan pair-comp-sound1 pair-comp-sound2 relcomp.simps*)
moreover have *AGTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$ = *pair-at-to-agtt* (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) (Δ-*eps-pair* $\mathcal{G}_1$ *?*Q_1 $\mathcal{G}_2$ *?*Q_2)
by (*auto simp*: *AGTT-comp-def pair-at-to-agtt-def gtt-interface-def* Δ_ε*-def' Δ-eps-pair-def*)
ultimately show *?thesis* **using** *pair-at-agtt-conv*[*of* Δ-*eps-pair* $\mathcal{G}_1$ *?*Q_1 $\mathcal{G}_2$ *?*Q_2 (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$)]
using *assms*
by (*auto simp*: Δ-*eps-pair-def gtt-states-def gtt-interface-def*)
qed

lemma *AGTT-comp'-sound*:
agtt-lang (*AGTT-comp'* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ *O agtt-lang* $\mathcal{G}_2$
using *AGTT-comp-sound*[*of fmap-states-gtt* (*Inl* :: *'b* ⇒ *'b* + *'c*) $\mathcal{G}_1$
fmap-states-gtt (*Inr* :: *'c* ⇒ *'b* + *'c*) $\mathcal{G}_2$]
by (*auto simp add*: *agtt-lang-fmap-states-gtt disjoint-iff-not-equal agtt-lang-Inl-Inr-states-agtt*)

4.11 Anchord gtt transitivity

definition *AGTT-trancl* :: (*'q*, *'f*) *gtt* ⇒ (*'q* + *'q*, *'f*) *gtt* **where**
AGTT-trancl $\mathcal{G}$ = (*let* $\mathcal{A}$ = *fmap-states-ta Inl* (*fst* $\mathcal{G}$) *in*
(*TA* (*rules* $\mathcal{A}$) (*eps* $\mathcal{A}$ |∪| *map-prod CInl CInr* |‘| (Δ-*Atrans-gtt* $\mathcal{G}$ (*fId-on* (*gtt-interface* $\mathcal{G}$)))),
TA (*map-ta-rule CInr id* |‘| (*rules* (*snd* $\mathcal{G}$))) (*map-both CInr* |‘| (*eps* (*snd* $\mathcal{G}$)))))

lemma *AGTT-trancl-sound*:
shows *agtt-lang* (*AGTT-trancl* $\mathcal{G}$) = (*agtt-lang* $\mathcal{G}$)$^+$
proof −
let *?P* = *map-prod* (*fmap-states-ta CInl*) (*fmap-states-ta CInr*) $\mathcal{G}$
let *?Q* = *fId-on* (*gtt-interface* $\mathcal{G}$) **let** *?Q'* = *map-prod CInl CInr* |‘| *?Q*
have *inv*: *finj-on CInl* ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) *finj-on CInr* ($\mathcal{Q}$ (*snd* $\mathcal{G}$))
?Q |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)

by (*auto simp*: *gtt-interface-def finj-CInl-CInr*)
have ∗: *fst* |‘| *map-prod CInl CInr* |‘| Δ-*Atrans-gtt* $\mathcal{G}$ (*fId-on* (*gtt-interface* $\mathcal{G}$)) |⊆| *CInl* |‘| $\mathcal{Q}$ (*fst* $\mathcal{G}$)
using *fsubsetD*[*OF* Δ-*Atrans-states-stable*[*OF inv*(*3*)]]
by (*auto simp add*: *gtt-interface-def*)
from *pair-at-lang-fun-states*[*OF inv*]
have *agtt-lang* $\mathcal{G}$ = *pair-at-lang ?P ?Q′* **using** *pair-at-agtt*[*of* $\mathcal{G}$] **by** *auto*
moreover then have (*agtt-lang* $\mathcal{G}$)$^+$ = *pair-at-lang ?P* (Δ-*Atrans-gtt ?P ?Q′*)
by (*simp add*: *pair-trancl-sound*)
moreover have *AGTT-trancl* $\mathcal{G}$ = *pair-at-to-agtt ?P* (Δ-*Atrans-gtt ?P ?Q′*)
using Δ-*Atrans-states-stable*[*OF inv*(*3*)] Δ-*Atrans-map-prod*[*OF inv*, *symmetric*]
using *fId-on-frelcomp-id*[*OF* ∗]
by (*auto simp*: *AGTT-trancl-def pair-at-to-agtt-def gtt-interface-def Let-def fmap-states-ta-def*)
(*metis fmap-prod-fimageI fmap-states fmap-states-ta-def*)
moreover have *gtt-interface* (*map-prod* (*fmap-states-ta CInl*) (*fmap-states-ta CInr*) $\mathcal{G}$) = {||}
by (*auto simp*: *gtt-interface-def*)
ultimately show *?thesis* **using** *pair-at-agtt-conv*[*of* Δ-*Atrans-gtt ?P ?Q′ ?P*] Δ-*Atrans-states-stable*[*OF inv*(*3*)]
unfolding Δ-*Atrans-map-prod*[*OF inv*, *symmetric*]
by (*simp add*: *fimage-mono gtt-interface-def map-prod-ftimes*)
qed

4.12 Anchord gtt intersection

definition *AGTT-inter* **where**
AGTT-inter $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ (*prod-ta* (*fst* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$),
prod-ta (*snd* $\mathcal{G}_1$) (*snd* $\mathcal{G}_2$))

lemma *AGTT-inter-sound*:
agtt-lang (*AGTT-inter* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ ∩ *agtt-lang* $\mathcal{G}_2$ (**is** *?Ls* = *?Rs*)
proof −
let *?TA-A* = *prod-ta* (*fst* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$)
let *?TA-B* = *prod-ta* (*snd* $\mathcal{G}_1$) (*snd* $\mathcal{G}_2$)
{fix *s t* **assume** *ass*: (*s*, *t*) ∈ *agtt-lang* (*AGTT-inter* $\mathcal{G}_1$ $\mathcal{G}_2$)
then obtain *q* **where** *ls*: *q* |∈| *ta-der ?TA-A* (*term-of-gterm s*) **and**
rs: *q* |∈| *ta-der ?TA-B* (*term-of-gterm t*)
by (*auto simp add*: *AGTT-inter-def agtt-lang-def gta-der-def*)
then have (*s*, *t*) ∈ *agtt-lang* $\mathcal{G}_1$ ∧ (*s*, *t*) ∈ *agtt-lang* $\mathcal{G}_2$
using *prod-ta-der-to-*$\mathcal{A}$-$\mathcal{B}$-*der1*[*of q*] *prod-ta-der-to-*$\mathcal{A}$-$\mathcal{B}$-*der2*[*of q*]
by (*auto simp*: *agtt-lang-def gta-der-def*) *blast***}**
then have *f*: *?Ls* ⊆ *?Rs* **by** *auto*
moreover have *?Rs* ⊆ *?Ls* **using** $\mathcal{A}$-$\mathcal{B}$-*der-to-prod-ta*
by (*fastforce simp*: *agtt-lang-def AGTT-inter-def gta-der-def*)
ultimately show *?thesis* **by** *blast*
qed

4.13 Anchord gtt triming

abbreviation *trim-agtt* ≡ *trim-gtt*

lemma *agtt-only-prod-lang*:
agtt-lang (*gtt-only-prod* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
proof −
let *?A* = *fst* $\mathcal{G}$ **let** *?B* = *snd* $\mathcal{G}$
have *?Ls* ⊆ *?Rs* **unfolding** *agtt-lang-def gtt-only-prod-def*
by (*auto simp*: *Let-def gta-der-def dest*: *ta-der-ta-only-prod-ta-der*)
moreover
{fix *s t* **assume** (*s*, *t*) ∈ *?Rs*
then obtain *q* **where** *r*: *q* |∈| *ta-der* (*fst* $\mathcal{G}$) (*term-of-gterm s*) *q* |∈| *ta-der* (*snd* $\mathcal{G}$) (*term-of-gterm t*)
by (*auto simp*: *agtt-lang-def gta-der-def*)
then have *q* |∈| *gtt-interface* $\mathcal{G}$ **by** (*auto simp*: *gtt-interface-def*)
then have (*s*, *t*) ∈ *?Ls* **using** *r*
by (*auto simp*: *agtt-lang-def gta-der-def gtt-only-prod-def Let-def intro*!: *exI*[*of - q*] *ta-der-only-prod ta-productive-setI*)**}**
ultimately show *?thesis* **by** *auto*
qed

lemma *agtt-only-reach-lang*:
agtt-lang (*gtt-only-reach* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
unfolding *agtt-lang-def gtt-only-reach-def*
by (*auto simp*: *gta-der-def simp flip*: *ta-der-gterm-only-reach*)

lemma *trim-agtt-lang* [*simp*]:
agtt-lang (*trim-agtt G*) = *agtt-lang G*
unfolding *trim-gtt-def comp-def agtt-only-prod-lang agtt-only-reach-lang* **..**

end
theory *RRn-Automata*
imports *Tree-Automata-Complement Ground-Ctxt*
begin

5 Regular relations

5.1 Encoding pairs of terms

The encoding of two terms s and t is given by its tree domain, which is the union of the domains of s and t, and the labels, which arise from looking up each position in s and t, respectively.

definition *gpair* :: *'f gterm* ⇒ *'g gterm* ⇒ (*'f option* × *'g option*) *gterm* **where**
gpair s t = *glabel* (λ*p*. (*gfun-at s p*, *gfun-at t p*)) (*gunion* (*gdomain s*) (*gdomain t*))

We provide an efficient implementation of gpair.

definition *zip-fill* :: *'a list* ⇒ *'b list* ⇒ (*'a option* × *'b option*) *list* **where**
zip-fill xs ys = *zip* (*map Some xs* @ *replicate* (*length ys* − *length xs*) *None*)
(*map Some ys* @ *replicate* (*length xs* − *length ys*) *None*)

lemma *zip-fill-code* [*code*]:
zip-fill xs [] = *map* (λ*x*. (*Some x*, *None*)) *xs*
zip-fill [] *ys* = *map* (λ*y*. (*None*, *Some y*)) *ys*
zip-fill (*x* # *xs*) (*y* # *ys*) = (*Some x*, *Some y*) # *zip-fill xs ys*
subgoal by (*induct xs*) (*auto simp*: *zip-fill-def*)
subgoal by (*induct ys*) (*auto simp*: *zip-fill-def*)
subgoal by (*auto simp*: *zip-fill-def*)
done

lemma *length-zip-fill* [*simp*]:
length (*zip-fill xs ys*) = *max* (*length xs*) (*length ys*)
by (*auto simp*: *zip-fill-def*)

lemma *nth-zip-fill*:
assumes *i* < *max* (*length xs*) (*length ys*)
shows *zip-fill xs ys* ! *i* = (*if i* < *length xs then Some* (*xs* ! *i*) *else None*, *if i* < *length ys then Some* (*ys* ! *i*) *else None*)
using *assms* **by** (*auto simp*: *zip-fill-def nth-append*)

fun *gpair-impl* :: *'f gterm option* ⇒ *'g gterm option* ⇒ (*'f option* × *'g option*) *gterm* **where**
gpair-impl (*Some s*) (*Some t*) = *gpair s t*
| *gpair-impl* (*Some s*) *None* = *map-gterm* (λ*f*. (*Some f*, *None*)) *s*
| *gpair-impl None* (*Some t*) = *map-gterm* (λ*f*. (*None*, *Some f*)) *t*
| *gpair-impl None* *None* = *GFun* (*None*, *None*) []

declare *gpair-impl.simps*(*2*−*4*)[*code*]

lemma *gpair-impl-code* [*simp*, *code*]:
gpair-impl (*Some s*) (*Some t*) =
(*case s of GFun f ss* ⇒ *case t of GFun g ts* ⇒
GFun (*Some f*, *Some g*) (*map* (λ(*s*, *t*). *gpair-impl s t*) (*zip-fill ss ts*)))
proof (*induct gdomain s gdomain t arbitrary*: *s t rule*: *gunion.induct*)
case (*1 f ss g ts*)
obtain *f' ss'* **where** [*simp*]: *s* = *GFun f' ss'* **by** (*cases s*)
obtain *g' ts'* **where** [*simp*]: *t* = *GFun g' ts'* **by** (*cases t*)
show *?case* **using** *1*(*2*,*3*) *1*(*1*)[*of i ss'* ! *i ts'* ! *i* **for** *i*]
by (*auto simp*: *gpair-def comp-def nth-zip-fill intro*: *glabel-map-gterm-conv*[*unfolded comp-def*]
intro!: *nth-equalityI*)
qed

lemma *gpair-code* [*code*]:
gpair s t = *gpair-impl* (*Some s*) (*Some t*)
by *simp*

declare *gpair-impl.simps(1)*[*simp del*]

We can easily prove some basic properties. I believe that proving them by induction with a definition along the lines of *gpair-impl* would be very cumbersome.

lemma *gpair-swap*:
map-gterm prod.swap (*gpair s t*) = *gpair t s*
by (*intro eq-gterm-by-gposs-gfun-at*) (*auto simp*: *gpair-def*)

lemma *gpair-assoc*:
defines $f \equiv \lambda(f, gh).\ (f, gh \gg\!= fst, gh \gg\!= snd)$
defines $g \equiv \lambda(fg, h).\ (fg \gg\!= fst, fg \gg\!= snd, h)$
shows *map-gterm f* (*gpair s* (*gpair t u*)) = *map-gterm g* (*gpair* (*gpair s t*) *u*)
by (*intro eq-gterm-by-gposs-gfun-at*) (*auto simp*: *gpair-def f-def g-def*)

5.2 Decoding of pairs

fun *gcollapse* :: *'f option gterm* ⇒ *'f gterm option* **where**
gcollapse (*GFun None* -) = *None*
| *gcollapse* (*GFun* (*Some f*) *ts*) = *Some* (*GFun f* (*map the* (*filter* (λ*t*. ¬ *Option.is-none t*) (*map gcollapse ts*))))

lemma *gcollapse-groot-None* [*simp*]:
groot-sym t = *None* ⟹ *gcollapse t* = *None*
fst (*groot t*) = *None* ⟹ *gcollapse t* = *None*
by (*cases t*, *simp*)+

definition *gfst* :: (*'f option* × *'g option*) *gterm* ⇒ *'f gterm* **where**
gfst = *the* ∘ *gcollapse* ∘ *map-gterm fst*

definition *gsnd* :: (*'f option* × *'g option*) *gterm* ⇒ *'g gterm* **where**
gsnd = *the* ∘ *gcollapse* ∘ *map-gterm snd*

lemma *filter-less-upt*:
[*i*←[*i*..<*m*] . *i* < *n*] = [*i*..<*min n m*]
proof (*cases i* ≤ *m*)
case *True* **then show** *?thesis*
proof (*induct rule*: *inc-induct*)
case (*step n*) **then show** *?case* **by** (*auto simp*: *upt-rec*[*of n*])
qed *simp*
qed *simp*

lemma *gcollapse-aux*:
assumes *gposs s* = {*p*. *p* ∈ *gposs t* ∧ *gfun-at t p* ≠ *Some None*}
shows *gposs* (*the* (*gcollapse t*)) = *gposs s*
⋀*p*. *p* ∈ *gposs s* ⟹ *gfun-at* (*the* (*gcollapse t*)) *p* = (*gfun-at t p* ⟜ *id*)

proof (*goal-cases*)
define s' t' **where** $s' \equiv$ *gdomain* s **and** $t' \equiv$ *gdomain* t
have $*$: *gposs* (*the* (*gcollapse* t)) = *gposs* $s \wedge$
($\forall p.\ p \in$ *gposs* $s \longrightarrow$ *gfun-at* (*the* (*gcollapse* t)) p = (*gfun-at* $t\ p \gg\!=$ *id*))
using *assms* s'*-def* t'*-def*
proof (*induct* s' t' *arbitrary*: s t *rule*: *gunion.induct*)
case (*1* f' ss' g' ts')
obtain f ss **where** s [*simp*]: s = *GFun* f ss **by** (*cases* s)
obtain g ts **where** t [*simp*]: t = *GFun* (*Some* g) ts
using *arg-cong*[*OF 1*(*2*), *of* $\lambda P.\ [] \in P$] **by** (*cases* t) *auto*
have $*$: $i <$ *length* $ts \Longrightarrow \neg$ *Option.is-none* (*gcollapse* (ts ! i)) $\longleftrightarrow i <$ *length* *ss* **for** i
using *arg-cong*[*OF 1*(*2*), *of* $\lambda P.\ [i] \in P$] **by** (*cases* ts ! i) *auto*
have l: *length* $ss \leq$ *length* ts
using *arg-cong*[*OF 1*(*2*), *of* $\lambda P.\ [$*length* $ss-1] \in P$] **by** *auto*
have [*simp*]: [t←*map gcollapse ts* . ¬ *Option.is-none* t] = *take* (*length* ss) (*map* *gcollapse* ts)
by (*subst* (*2*) *map-nth*[*symmetric*]) (*auto simp add*: $*$ *filter-map comp-def* *filter-less-upt*
cong: *filter-cong*[*OF refl*, *of* [*0*..<*length ts*], *unfolded set-upt atLeast-LessThan-iff*]
intro!: *nth-equalityI*)
have [*simp*]: $i <$ *length* $ss \Longrightarrow$ *gposs* (ss ! i) = *gposs* (*the* (*gcollapse* (ts ! i))) **for** i
using *conjunct1*[*OF 1*(*1*), *of i ss* ! *i ts* ! *i*] *l arg-cong*[*OF 1*(*2*), *of* $\lambda P.$ {$p.\ i$ # $p \in P$}]
1(*3*,*4*) **by** *simp*
show *?case*
proof (*intro conjI allI*, *goal-cases A B*)
case *A* **show** *?case* **using** l **by** (*auto simp*: *comp-def split*: *if-splits*)
next
case (*B* p) **show** *?case*
proof (*cases* p)
case (*Cons* i q) **then show** *?thesis* **using** *arg-cong*[*OF 1*(*2*), *of* $\lambda P.$ {$p.\ i$ # $p \in P$}]
spec[*OF conjunct2*[*OF 1*(*1*)], *of i ss* ! *i ts* ! *i q*] *1*(*3*,*4*) **by** *auto*
qed *auto*
qed
qed
{ case *1* **then show** *?case* **using** $*$ **by** *blast*
next
case *2* **then show** *?case* **using** $*$ **by** *blast* **}**
qed

lemma *gfst-gpair*:
gfst (*gpair* s t) = s
proof −
have $*$: *gposs* s = {$p \in$ *gposs* (*map-gterm fst* (*gpair* s t)). *gfun-at* (*map-gterm fst* (*gpair* s t)) $p \neq$ *Some None*}

using *gfun-at-nongposs*
by (*fastforce simp*: *gpair-def elim*: *gfun-at-possE*)
show *?thesis* **unfolding** *gfst-def comp-def* **using** *gcollapse-aux*[*OF* ∗]
by (*auto intro*!: *eq-gterm-by-gposs-gfun-at simp*: *gpair-def*)
qed

lemma *gsnd-gpair*:
gsnd (*gpair s t*) = *t*
using *gfst-gpair*[*of t s*] *gpair-swap*[*of t s*, *symmetric*]
by (*simp add*: *gfst-def gsnd-def gpair-def gterm.map-comp comp-def*)

lemma *gpair-impl-None-Inv*:
map-gterm (*the* ∘ *snd*) (*gpair-impl None* (*Some t*)) = *t*
by (*simp add*: *gterm.map-ident gterm.map-comp comp-def*)

5.3 Contexts to gpair

lemma *gpair-context1*:
assumes *length ts* = *length us*
shows *gpair* (*GFun f ts*) (*GFun f us*) = *GFun* (*Some f*, *Some f*) (*map* (*case-prod gpair*) (*zip ts us*))
using *assms* **unfolding** *gpair-code* **by** (*auto intro*!: *nth-equalityI simp*: *zip-fill-def*)

lemma *gpair-context2*:
assumes ⋀ *i*. *i* < *length ts* ⟹ *ts* ! *i* = *gpair* (*ss* ! *i*) (*us* ! *i*)
and *length ss* = *length ts* **and** *length us* = *length ts*
shows *GFun* (*Some f*, *Some h*) *ts* = *gpair* (*GFun f ss*) (*GFun h us*)
using *assms* **unfolding** *gpair-code gpair-impl-code*
by (*auto simp*: *zip-fill-def intro*!: *nth-equalityI*)

lemma *map-funs-term-some-gpair*:
shows *gpair t t* = *map-gterm* (λ*f*. (*Some f*, *Some f*)) *t*
proof (*induct t*)
case (*GFun f ts*)
then show *?case* **by** (*auto intro*!: *gpair-context2*[*symmetric*])
qed

lemma *gpair-inject* [*simp*]:
gpair s t = *gpair s′ t′* ⟷ *s* = *s′* ∧ *t* = *t′*
by (*metis gfst-gpair gsnd-gpair*)

abbreviation *gterm-to-None-Some* :: *′f gterm* ⇒ (*′f option* × *′f option*) *gterm*
where
gterm-to-None-Some t ≡ *map-gterm* (λ*f*. (*None*, *Some f*)) *t*
abbreviation *gterm-to-Some-None t* ≡ *map-gterm* (λ*f*. (*Some f*, *None*)) *t*

lemma *inj-gterm-to-None-Some*: *inj gterm-to-None-Some*
by (*meson Pair-inject gterm.inj-map inj-onI option.inject*)

lemma *zip-fill1*:
assumes *length ss* < *length ts*
shows *zip-fill ss ts* = *zip* (*map Some ss*) (*map Some* (*take* (*length ss*) *ts*)) @
map (λ *x*. (*None*, *Some x*)) (*drop* (*length ss*) *ts*)
using *assms* **by** (*auto simp*: *zip-fill-def list-eq-iff-nth-eq nth-append simp add*: *min.absorb2*)

lemma *zip-fill2*:
assumes *length ts* < *length ss*
shows *zip-fill ss ts* = *zip* (*map Some* (*take* (*length ts*) *ss*)) (*map Some ts*) @
map (λ *x*. (*Some x*, *None*)) (*drop* (*length ts*) *ss*)
using *assms* **by** (*auto simp*: *zip-fill-def list-eq-iff-nth-eq nth-append simp add*: *min.absorb2*)

lemma *not-gposs-append* [*simp*]:
assumes *p* ∉ *gposs t*
shows *p* @ *q* ∈ *gposs t* = *False* **using** *assms poss-gposs-conv*
using *poss-append-poss* **by** *blast*

lemma *gfun-at-gpair*:
gfun-at (*gpair s t*) *p* = (*if p* ∈ *gposs s then* (*if p* ∈ *gposs t*
then Some (*gfun-at s p*, *gfun-at t p*)
else Some (*gfun-at s p*, *None*)) *else*
(*if p* ∈ *gposs t then Some* (*None*, *gfun-at t p*) *else None*))
using *gfun-at-glabel* **by** (*auto simp*: *gpair-def*)

lemma *gposs-of-gpair* [*simp*]:
shows *gposs* (*gpair s t*) = *gposs s* ∪ *gposs t*
by (*auto simp*: *gpair-def*)

lemma *poss-to-gpair-poss*:
p ∈ *gposs s* ⟹ *p* ∈ *gposs* (*gpair s t*)
p ∈ *gposs t* ⟹ *p* ∈ *gposs* (*gpair s t*)
by *auto*

lemma *gsubt-at-gpair-poss*:
assumes *p* ∈ *gposs s* **and** *p* ∈ *gposs t*
shows *gsubt-at* (*gpair s t*) *p* = *gpair* (*gsubt-at s p*) (*gsubt-at t p*) **using** *assms*
by (*auto simp*: *gunion-gsubt-at-poss gfun-at-gpair intro*!: *eq-gterm-by-gposs-gfun-at*)

lemma *subst-at-gpair-nt-poss-Some-None*:
assumes *p* ∈ *gposs s* **and** *p* ∉ *gposs t*
shows *gsubt-at* (*gpair s t*) *p* = *gterm-to-Some-None* (*gsubt-at s p*) **using** *assms*

gfun-at-poss
by (*force simp*: *gunion-gsubt-at-poss gfun-at-gpair intro*!: *eq-gterm-by-gposs-gfun-at*)

lemma *subst-at-gpair-nt-poss-None-Some*:
assumes $p \in gposs\ t$ **and** $p \notin gposs\ s$
shows *gsubt-at* (*gpair s t*) *p* = *gterm-to-None-Some* (*gsubt-at t p*) **using** *assms*
gfun-at-poss
by (*force simp*: *gunion-gsubt-at-poss gfun-at-gpair intro*!: *eq-gterm-by-gposs-gfun-at*)

lemma *gpair-ctxt-decomposition*:
fixes *C* **defines** $p \equiv ghole\text{-}pos\ C$
assumes $p \notin gposs\ s$ **and** $gpair\ s\ t = C\langle gterm\text{-}to\text{-}None\text{-}Some\ u\rangle_G$
shows $gpair\ s\ (gctxt\text{-}at\text{-}pos\ t\ p)\langle v\rangle_G = C\langle gterm\text{-}to\text{-}None\text{-}Some\ v\rangle_G$
using *assms*(2−)
proof −
note *p*[*simp*] = *assms*(1)
have *pt*: $p \in gposs\ t$ **and** *pc*: $p \in gposs\ C\langle gterm\text{-}to\text{-}None\text{-}Some\ v\rangle_G$
and *pu*: $p \in gposs\ C\langle gterm\text{-}to\text{-}None\text{-}Some\ u\rangle_G$
using *arg-cong*[*OF assms*(3), *of gposs*] *assms*(2) *ghole-pos-in-apply*
by *auto*
have ∗: *gctxt-at-pos* (*gpair s* (*gctxt-at-pos t* (*ghole-pos C*))⟨*v*⟩$_G$) (*ghole-pos C*) =
gctxt-at-pos (*gpair s t*) (*ghole-pos C*)
using *assms*(2) *pt*
by (*intro eq-gctxt-at-pos*)
(*auto simp*: *gposs-gctxt-at-pos gunion-gsubt-at-poss gfun-at-gpair gfun-at-gctxt-at-pos-not-after*)
have *gsubt-at* (*gpair s* (*gctxt-at-pos t p*)⟨*v*⟩$_G$) *p* = *gsubt-at C*⟨*gterm-to-None-Some*
v⟩$_G$ *p*
using *pt assms*(2) *subst-at-gpair-nt-poss-None-Some*[*OF* - *assms*(2), *of* (*gctxt-at-pos*
t p)⟨*v*⟩$_G$]
using *ghole-pos-gctxt-at-pos*
by (*simp add*: *ghole-pos-in-apply*)
then show *?thesis* **using** *assms*(2) *ghole-pos-gctxt-at-pos*
using *gsubst-at-gctxt-at-eq-gtermD*[*OF assms*(3) *pu*]
by (*intro gsubst-at-gctxt-at-eq-gtermI*[*OF* - *pc*])
(*auto simp*: *ghole-pos-in-apply* ∗ *gposs-gctxt-at-pos*[*OF pt*, *unfolded p*])
qed

lemma *groot-gpair* [*simp*]:
fst (*groot* (*gpair s t*)) = (*Some* (*fst* (*groot s*)), *Some* (*fst* (*groot t*)))
by (*cases s*; *cases t*) (*auto simp add*: *gpair-code*)

lemma *ground-ctxt-adapt-ground* [*intro*]:
assumes *ground-ctxt C*
shows *ground-ctxt* (*adapt-vars-ctxt C*)
using *assms* **by** (*induct C*) *auto*

lemma *adapt-vars-ctxt2* :
assumes *ground-ctxt C*

shows *adapt-vars-ctxt* (*adapt-vars-ctxt C*) = *adapt-vars-ctxt C* **using** *assms*
by (*induct C*) (*auto simp*: *adapt-vars2*)

5.4 Encoding of lists of terms

definition *gencode* :: *'f gterm list* ⇒ *'f option list gterm* **where**
gencode ts = *glabel* (λ*p*. *map* (λ*t*. *gfun-at t p*) *ts*) (*gunions* (*map gdomain ts*))

definition *gdecode-nth* :: *'f option list gterm* ⇒ *nat* ⇒ *'f gterm* **where**
gdecode-nth t i = *the* (*gcollapse* (*map-gterm* (λ*f*. *f* ! *i*) *t*))

lemma *gdecode-nth-gencode*:
assumes *i* < *length ts*
shows *gdecode-nth* (*gencode ts*) *i* = *ts* ! *i*
proof −
have ∗: *gposs* (*ts* ! *i*) = {*p* ∈ *gposs* (*map-gterm* (λ*f*. *f* ! *i*) (*gencode ts*)).
gfun-at (*map-gterm* (λ*f*. *f* ! *i*) (*gencode ts*)) *p* ≠ *Some None*}
using *assms*
by (*auto simp*: *gencode-def elim*: *gfun-at-possE dest*: *gfun-at-poss-gpossD*) (*force simp*: *fun-at-def' split*: *if-splits*)
show *?thesis* **unfolding** *gdecode-nth-def comp-def* **using** *assms gcollapse-aux*[*OF* ∗]
by (*auto intro*!: *eq-gterm-by-gposs-gfun-at simp*: *gencode-def*)
(*metis* (*no-types*) *gposs-map-gterm length-map list.set-map map-nth-eq-conv nth-mem*)
qed

definition *gdecode* :: *'f option list gterm* ⇒ *'f gterm list* **where**
gdecode t = (*case t of GFun f ts* ⇒ *map* (λ*i*. *gdecode-nth t i*) [*0*..<*length f*])

lemma *gdecode-gencode*:
gdecode (*gencode ts*) = *ts*
proof (*cases gencode ts*)
case (*GFun f ts'*)
have *length f* = *length ts* **using** *arg-cong*[*OF GFun*, *of* λ*t*. *gfun-at t* []]
by (*auto simp*: *gencode-def*)
then show *?thesis* **using** *gdecode-nth-gencode*[*of - ts*]
by (*auto intro*!: *nth-equalityI simp*: *gdecode-def GFun*)
qed

definition *gencode-impl* :: *'f gterm option list* ⇒ *'f option list gterm* **where**
gencode-impl ts = *glabel* (λ*p*. *map* (λ*t*. *t* ⤜ (λ*t*. *gfun-at t p*)) *ts*) (*gunions* (*map* (*case-option* (*GFun* () []) *gdomain*) *ts*))

lemma *gencode-code* [*code*]:
gencode ts = *gencode-impl* (*map Some ts*)
by (*auto simp*: *gencode-def gencode-impl-def comp-def*)

lemma *gencode-singleton*:

gencode [*t*] = *map-gterm* (λ*f*. [*Some f*]) *t*
using *glabel-map-gterm-conv*[*unfolded comp-def*, *of* λ*t*. [*t*] *t*]
by (*simp add*: *gunions-def gencode-def*)

lemma *gencode-pair*:
gencode [*t*, *u*] = *map-gterm* (λ(*f*, *g*). [*f*, *g*]) (*gpair t u*)
by (*simp add*: *gunions-def gencode-def gpair-def map-gterm-glabel comp-def*)

5.5 RRn relations

definition *RR1-spec* **where**
RR1-spec A T ⟷ *ℒ A* = *T*

definition *RR2-spec* **where**
RR2-spec A T ⟷ *ℒ A* = {*gpair t u* |*t u*. (*t*, *u*) ∈ *T*}

definition *RRn-spec* **where**
RRn-spec n A R ⟷ *ℒ A* = *gencode* ‘ *R* ∧ (∀ *ts* ∈ *R*. *length ts* = *n*)

lemma *RR1-to-RRn-spec*:
assumes *RR1-spec A T*
shows *RRn-spec 1* (*fmap-funs-reg* (λ*f*. [*Some f*]) *A*) ((λ*t*. [*t*]) ‘ *T*)
proof −
have [*simp*]: *inj-on* (λ*f*. [*Some f*]) *X* **for** *X* **by** (*auto simp*: *inj-on-def*)
show *?thesis* **using** *assms*
by (*auto simp*: *RR1-spec-def RRn-spec-def fmap-funs-ℒ image-comp comp-def gencode-singleton*)
qed

lemma *RR2-to-RRn-spec*:
assumes *RR2-spec A T*
shows *RRn-spec 2* (*fmap-funs-reg* (λ(*f*, *g*). [*f*, *g*]) *A*) ((λ(*t*, *u*). [*t*, *u*]) ‘ *T*)
proof −
have [*simp*]: *inj-on* (λ(*f*, *g*). [*f*, *g*]) *X* **for** *X* **by** (*auto simp*: *inj-on-def*)
show *?thesis* **using** *assms*
by (*auto simp*: *RR2-spec-def RRn-spec-def fmap-funs-ℒ image-comp comp-def prod.case-distrib gencode-pair*)
qed

lemma *RRn-to-RR2-spec*:
assumes *RRn-spec 2 A T*
shows *RR2-spec* (*fmap-funs-reg* (λ *f*. (*f* ! *0* , *f* ! *1*)) *A*) ((λ *f*. (*f* ! *0*, *f* ! *1*)) ‘ *T*) (**is** *RR2-spec ?A ?T*)
proof −
{fix *xs* **assume** *xs* ∈ *T* **then have** *length xs* = *2* **using** *assms* **by** (*auto simp*: *RRn-spec-def*)
then obtain *t u* **where** ∗: *xs* = [*t*, *u*]
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-1 length-0-conv length-Suc-conv*)
have ∗∗: (λ*f*. (*f* ! *0*, *f* ! *Suc 0*)) ∘ (λ(*f*, *g*). [*f*, *g*]) = *id* **by** *auto*

have *map-gterm* (λ*f*. (*f* ! *0*, *f* ! *Suc 0*)) (*gencode xs*) = *gpair t u*
unfolding ∗ *gencode-pair gterm.map-comp* ∗∗ *gterm.map-id* **..**
then have ∃ *t u*. *xs* = [*t*, *u*] ∧ *map-gterm* (λ*f*. (*f* ! *0*, *f* ! *Suc 0*)) (*gencode xs*) = *gpair t u*
using ∗ **by** *blast***}**
then show *?thesis* **using** *assms*
by (*force simp*: *RR2-spec-def RRn-spec-def fmap-funs-*$\mathcal{L}$ *image-comp comp-def prod.case-distrib gencode-pair image-iff Bex-def*)
qed

lemma *relabel-RR1-spec* [*simp*]:
RR1-spec (*relabel-reg A*) *T* ⟷ *RR1-spec A T*
by (*simp add*: *RR1-spec-def*)

lemma *relabel-RR2-spec* [*simp*]:
RR2-spec (*relabel-reg A*) *T* ⟷ *RR2-spec A T*
by (*simp add*: *RR2-spec-def*)

lemma *relabel-RRn-spec* [*simp*]:
RRn-spec n (*relabel-reg A*) *T* ⟷ *RRn-spec n A T*
by (*simp add*: *RRn-spec-def*)

lemma *trim-RR1-spec* [*simp*]:
RR1-spec (*trim-reg A*) *T* ⟷ *RR1-spec A T*
by (*simp add*: *RR1-spec-def* $\mathcal{L}$*-trim*)

lemma *trim-RR2-spec* [*simp*]:
RR2-spec (*trim-reg A*) *T* ⟷ *RR2-spec A T*
by (*simp add*: *RR2-spec-def* $\mathcal{L}$*-trim*)

lemma *trim-RRn-spec* [*simp*]:
RRn-spec n (*trim-reg A*) *T* ⟷ *RRn-spec n A T*
by (*simp add*: *RRn-spec-def* $\mathcal{L}$*-trim*)

lemma *swap-RR2-spec*:
assumes *RR2-spec A R*
shows *RR2-spec* (*fmap-funs-reg prod.swap A*) (*prod.swap* ‘ *R*) **using** *assms*
by (*force simp add*: *RR2-spec-def fmap-funs-*$\mathcal{L}$ *gpair-swap image-iff*)

5.6 Nullary automata

lemma *false-RRn-spec*:
RRn-spec n empty-reg {}
by (*auto simp*: *RRn-spec-def* $\mathcal{L}$*-epmty*)

lemma *true-RR0-spec*:
RRn-spec 0 (*Reg* {|*q*|} (*TA* {|[] [] → *q*|} {||})) {[]}
by (*auto simp*: *RRn-spec-def* $\mathcal{L}$*-def const-ta-lang gencode-def gunions-def*)

5.7 Pairing RR1 languages

cf. *gpair*.

abbreviation *lift-Some-None s* ≡ (*Some s*, *None*)
abbreviation *lift-None-Some s* ≡ (*None*, *Some s*)
abbreviation *pair-eps A B* ≡ (λ (*p*, *q*). ((*Some* (*fst p*), *q*), (*Some* (*snd p*), *q*))) |‘| (*eps A* |×| *finsert None* (*Some* |‘| 𝒬 *B*))
abbreviation *pair-rule* ≡ (λ (*ra*, *rb*). *TA-rule* (*Some* (*r-root ra*), *Some* (*r-root rb*)) (*zip-fill* (*r-lhs-states ra*) (*r-lhs-states rb*)) (*Some* (*r-rhs ra*), *Some* (*r-rhs rb*)))

lemma *lift-Some-None-pord-swap* [*simp*]:
prod.swap ∘ *lift-Some-None* = *lift-None-Some*
prod.swap ∘ *lift-None-Some* = *lift-Some-None*
by *auto*

lemma *eps-to-pair-eps-Some-None*:
(*p*, *q*) |∈| *eps* 𝒜 ⟹ (*lift-Some-None p*, *lift-Some-None q*) |∈| *pair-eps* 𝒜 ℬ
by *force*

definition *pair-automaton* :: (′*p*, ′*f*) *ta* ⇒ (′*q*, ′*g*) *ta* ⇒ (′*p option* × ′*q option*, ′*f option* × ′*g option*) *ta* **where**
pair-automaton A B = *TA*
(*map-ta-rule lift-Some-None lift-Some-None* |‘| *rules A* |∪|
map-ta-rule lift-None-Some lift-None-Some |‘| *rules B* |∪|
pair-rule |‘| (*rules A* |×| *rules B*))
(*pair-eps A B* |∪| *map-both prod.swap* |‘| (*pair-eps B A*))

definition *pair-automaton-reg* **where**
pair-automaton-reg R L = *Reg* (*Some* |‘| *fin R* |×| *Some* |‘| *fin L*) (*pair-automaton* (*ta R*) (*ta L*))

lemma *pair-automaton-eps-simps*:
(*lift-Some-None p*, *p*′) |∈| *eps* (*pair-automaton A B*) ⟷ (*lift-Some-None p*, *p*′) |∈| *pair-eps A B*
(*q* , *lift-Some-None q*′) |∈| *eps* (*pair-automaton A B*) ⟷ (*q* , *lift-Some-None q*′) |∈| *pair-eps A B*
by (*auto simp*: *pair-automaton-def eps-to-pair-eps-Some-None*)

lemma *pair-automaton-eps-Some-SomeD*:
((*Some p*, *Some p*′), *r*) |∈| *eps* (*pair-automaton A B*) ⟹ *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧ (*Some p* = *fst r* ∨ *Some p*′ = *snd r*) ∧
(*Some p* ≠ *fst r* ⟶ (*p*, *the* (*fst r*)) |∈| (*eps A*)) ∧ (*Some p*′ ≠ *snd r* ⟶ (*p*′, *the* (*snd r*)) |∈| (*eps B*))
by (*auto simp*: *pair-automaton-def*)

lemma *pair-automaton-eps-Some-SomeD2*:
(*r*, (*Some p*, *Some p*′)) |∈| *eps* (*pair-automaton A B*) ⟹ *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧ (*fst r* = *Some p* ∨ *snd r* = *Some p*′) ∧

(*fst r* ≠ *Some p* ⟶ (*the* (*fst r*), *p*) |∈| (*eps A*)) ∧ (*snd r* ≠ *Some p′* ⟶ (*the* (*snd r*), *p′*) |∈| (*eps B*))
by (*auto simp*: *pair-automaton-def*)

lemma *pair-eps-Some-None*:
fixes *p q q′*
defines *l* ≡ (*p*, *q*) **and** *r* ≡ *lift-Some-None q′*
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))|⁺|
shows *q* = *None* ∧ *p* ≠ *None* ∧ (*the p*, *q′*) |∈| (*eps A*)|⁺| **using** *assms*(*3*, *1*, *2*)
proof (*induct arbitrary*: *q′ q rule*: *ftrancl-induct*)
case (*Step b*)
then show *?case* **unfolding** *pair-automaton-eps-simps*
by (*auto simp*: *pair-automaton-eps-simps*)
(*meson not-ftrancl-into*)
qed (*auto simp*: *pair-automaton-def*)

lemma *pair-eps-Some-Some*:
fixes *p q*
defines *l* ≡ (*Some p*, *Some q*)
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))|⁺|
shows *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧
(*fst l* ≠ *fst r* ⟶ (*p*, *the* (*fst r*)) |∈| (*eps A*)|⁺|) ∧
(*snd l* ≠ *snd r* ⟶ (*q*, *the* (*snd r*)) |∈| (*eps B*)|⁺|)
using *assms*(*2*, *1*)
proof (*induct arbitrary*: *p q rule*: *ftrancl-induct*)
case (*Step b c*)
then obtain *r r′* **where** ∗: *b* = (*Some r*, *Some r′*) **by** (*cases b*) *auto*
show *?case* **using** *Step*(*2*)
using *pair-automaton-eps-Some-SomeD*[*OF Step*(*3*)[*unfolded* ∗]]
by (*auto simp*: ∗) (*meson not-ftrancl-into*)+
qed (*auto simp*: *pair-automaton-def*)

lemma *pair-eps-Some-Some2*:
fixes *p q*
defines *r* ≡ (*Some p*, *Some q*)
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))|⁺|
shows *fst l* ≠ *None* ∧ *snd l* ≠ *None* ∧
(*fst l* ≠ *fst r* ⟶ (*the* (*fst l*), *p*) |∈| (*eps A*)|⁺|) ∧
(*snd l* ≠ *snd r* ⟶ (*the* (*snd l*), *q*) |∈| (*eps B*)|⁺|)
using *assms*(*2*, *1*)
proof (*induct arbitrary*: *p q rule*: *ftrancl-induct*)
case (*Step b c*)
from *pair-automaton-eps-Some-SomeD2*[*OF Step*(*3*)]
obtain *r r′* **where** ∗: *c* = (*Some r*, *Some r′*) **by** (*cases c*) *auto*
from *Step*(*2*)[*OF this*] **show** *?case*
using *pair-automaton-eps-Some-SomeD*[*OF Step*(*3*)[*unfolded* ∗]]
by (*auto simp*: ∗) (*meson not-ftrancl-into*)+
qed (*auto simp*: *pair-automaton-def*)

lemma *map-pair-automaton*:
pair-automaton (*fmap-funs-ta f A*) (*fmap-funs-ta g B*) =
fmap-funs-ta (λ(*a*, *b*). (*map-option f a*, *map-option g b*)) (*pair-automaton A B*)
(**is** *?Ls* = *?Rs*)
proof −
let *?ls* = *pair-rule* ∘ *map-prod* (*map-ta-rule id f*) (*map-ta-rule id g*)
let *?rs* = *map-ta-rule id* (λ(*a*, *b*). (*map-option f a*, *map-option g b*)) ∘ *pair-rule*
have ∗: (λ(*a*, *b*). (*map-option f a*, *map-option g b*)) ∘ *lift-Some-None* = *lift-Some-None* ∘ *f*
(λ(*a*, *b*). (*map-option f a*, *map-option g b*)) ∘ *lift-None-Some* = *lift-None-Some* ∘ *g*
by (*auto simp*: *comp-def*)
have *?ls x* = *?rs x* **for** *x*
by (*cases x*) (*auto simp*: *ta-rule.map-sel*)
then have [*simp*]: *?ls* = *?rs* **by** *blast*
then have *rules ?Ls* = *rules ?Rs*
unfolding *pair-automaton-def fmap-funs-ta-def*
by (*simp add*: *fimage-funion map-ta-rule-comp* ∗ *map-prod-ftimes*)
moreover have *eps ?Ls* = *eps ?Rs*
unfolding *pair-automaton-def fmap-funs-ta-def*
by (*simp add*: *fimage-funion* Q*-def*)
ultimately show *?thesis*
by (*intro TA-equalityI*) *simp*
qed

lemmas *map-pair-automaton-12* =
map-pair-automaton[*of* - - *id*, *unfolded fmap-funs-ta-id option.map-id*]
map-pair-automaton[*of id* - -, *unfolded fmap-funs-ta-id option.map-id*]

lemma *fmap-states-funs-ta-commute*:
fmap-states-ta f (*fmap-funs-ta g A*) = *fmap-funs-ta g* (*fmap-states-ta f A*)
proof −
have [*simp*]: *map-ta-rule f id* (*map-ta-rule id g r*) = *map-ta-rule id g* (*map-ta-rule f id r*) **for** *r*
by (*cases r*) *auto*
show *?thesis*
by (*auto simp*: *ta-rule.case-distrib fmap-states-ta-def fmap-funs-ta-def fimage-iff fBex-def split*: *ta-rule.splits*)
qed

lemma *states-pair-automaton*:
Q (*pair-automaton A B*) |⊆| (*finsert None* (*Some* |‘| Q *A*) |×| (*finsert None* (*Some* |‘| Q *B*)))
unfolding *pair-automaton-def*
apply (*intro* Q*-subseteq-I*)
apply (*auto simp*: *ta-rule.map-sel in-fset-conv-nth nth-zip-fill rule-statesD eps-statesD*)
apply (*simp add*: *rule-statesD*)+
done

lemma *swap-pair-automaton*:
assumes (*p*, *q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm t*)
shows (*q*, *p*) |∈| *ta-der* (*pair-automaton B A*) (*term-of-gterm* (*map-gterm prod.swap t*))
proof −
let *?m* = *map-ta-rule prod.swap prod.swap*
have [*simp*]: *map prod.swap* (*zip-fill xs ys*) = *zip-fill ys xs* **for** *xs ys*
by (*auto simp*: *zip-fill-def zip-nth-conv comp-def intro*!: *nth-equalityI*)
let *?swp* = λ*X*. *fmap-states-ta prod.swap* (*fmap-funs-ta prod.swap X*)
{ fix *A B*
let *?AB* = *?swp* (*pair-automaton A B*) **and** *?BA* = *pair-automaton B A*
have *eps ?AB* |⊆| *eps ?BA rules ?AB* |⊆| *rules ?BA*
by (*auto simp*: *fmap-states-ta-def fmap-funs-ta-def pair-automaton-def fimage-funion comp-def prod.map-comp ta-rule.map-comp*)
} note ∗ = *this*
let *?BA* = *?swp* (*?swp* (*pair-automaton B A*)) **and** *?AB* = *?swp* (*pair-automaton A B*)
have ∗∗: *r* |∈| *rules* (*pair-automaton B A*) ⟹ *?m r* |∈| *?m* |‘| *rules* (*pair-automaton B A*) **for** *r*
by *blast*
have *r* |∈| *rules ?BA* ⟹ *r* |∈| *rules ?AB e* |∈| *eps ?BA* ⟹ *e* |∈| *eps ?AB* **for** *r e*
using ∗[*of B A*] *map-ta-rule-prod-swap-id*
unfolding *fmap-funs-ta-def fmap-states-ta-def ta.sel*
by (*auto simp*: *map-ta-rule-comp image-iff ta-rule.map-id0 intro*!: *bexI*[*of - ?m r*])
then have *eps ?BA* |⊆| *eps ?AB rules ?BA* |⊆| *rules ?AB*
by *blast*+
then have *fmap-states-ta prod.swap* (*fmap-funs-ta prod.swap* (*pair-automaton A B*)) = *pair-automaton B A*
using ∗[*of A B*] **by** (*simp add*: *fmap-states-funs-ta-commute fmap-funs-ta-comp fmap-states-ta-comp TA-equalityI*)
then show *?thesis*
using *ta-der-fmap-states-ta*[*OF ta-der-fmap-funs-ta*[*OF assms*], *of prod.swap prod.swap*]
by (*simp add*: *gterm.map-comp comp-def*)
qed

lemma *to-ta-der-pair-automaton*:
p |∈| *ta-der A* (*term-of-gterm t*) ⟹
(*Some p*, *None*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*map-gterm* (λ*f*. (*Some f*, *None*)) *t*))
q |∈| *ta-der B* (*term-of-gterm u*) ⟹
(*None*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*map-gterm* (λ*f*. (*None*, *Some f*)) *u*))
p |∈| *ta-der A* (*term-of-gterm t*) ⟹ *q* |∈| *ta-der B* (*term-of-gterm u*) ⟹
(*Some p*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*gpair t u*))

```
proof (goal-cases sn ns ss)
  let ?AB = pair-automaton A B
 have 1: (Some p, None) |∈| ta-der (pair-automaton A B) (term-of-gterm (map-gterm
(λf. (Some f, None)) s))
    if p |∈| ta-der A (term-of-gterm s) for A B p s
  proof (rule fsubsetD[OF ta-der-mono])
    show rules (fmap-states-ta lift-Some-None (fmap-funs-ta lift-Some-None A))
|⊆|
     rules (pair-automaton A B)
     by (auto simp: fmap-states-ta-def fmap-funs-ta-def comp-def ta-rule.map-comp
          pair-automaton-def)
  next
   show eps (fmap-states-ta lift-Some-None (fmap-funs-ta lift-Some-None A)) |⊆|
     eps (pair-automaton A B)
     by (rule fsubsetI)
      (auto simp: fmap-states-ta-def fmap-funs-ta-def pair-automaton-def comp-def
fimage.rep-eq
         dest: eps-to-pair-eps-Some-None)
  next
   show lift-Some-None p |∈| ta-der
     (fmap-states-ta lift-Some-None (fmap-funs-ta lift-Some-None A))
     (term-of-gterm (gterm-to-Some-None s))
    using ta-der-fmap-states-ta[OF ta-der-fmap-funs-ta[OF that], of lift-Some-None]
     using ta-der-fmap-funs-ta ta-der-to-fmap-states-der that by fastforce
  qed
  have 2: q |∈| ta-der B (term-of-gterm t) ⟹
   (None, Some q) |∈| ta-der ?AB (term-of-gterm (map-gterm (λg. (None, Some
g)) t))
   for q t using swap-pair-automaton[OF 1[of q B t A]] by (simp add: gterm.map-comp
comp-def)
  {
   case sn then show ?case by (rule 1)
  next
   case ns then show ?case by (rule 2)
  next
   case ss then show ?case
   proof (induct t arbitrary: p q u)
    case (GFun f ts)
    obtain g us where u [simp]: u = GFun g us by (cases u)
    obtain p' ps where p': f ps → p' |∈| rules A p' = p ∨ (p', p) |∈| (eps A)|^+|
length ps = length ts
       ⋀i. i < length ts ⟹ ps ! i |∈| ta-der A (term-of-gterm (ts ! i)) using
GFun(2) by auto
     obtain q' qs where q': g qs → q' |∈| rules B q' = q ∨ (q', q) |∈| (eps B)|^+|
length qs = length us
       ⋀i. i < length us ⟹ qs ! i |∈| ta-der B (term-of-gterm (us ! i)) using
GFun(3) by auto
     have ∗: Some p |∈| Some |`| Q A Some q' |∈| Some |`| Q B
      using p'(2,1) q'(1)
```

```
      by (auto simp: rule-statesD eps-trancl-statesD)
    have eps: p' = p ∧ q' = q ∨ ((Some p', Some q'), (Some p, Some q)) |∈| (eps
?AB)|^+|
    proof (cases p' = p)
      case True note p = this then show ?thesis
      proof (cases q' = q)
        case False
        then have (q', q) |∈| (eps B)|^+| using q'(2) by auto
       hence ((Some p', Some q'), Some p', Some q) |∈| (eps (pair-automaton A
B))|^+|
        proof (rule ftrancl-map[rotated])
          fix x y
          assume (x, y) |∈| eps B
          thus ((Some p', Some x), Some p', Some y) |∈| eps (pair-automaton A
B)
            using p'(1)[THEN rule-statesD(1), simplified]
            apply (simp add: pair-automaton-def image-iff fSigma.rep-eq)
            by fastforce
        qed
        thus ?thesis
          by (simp add: p)
      qed (simp add: p)
    next
      case False note p = this
      then have ∗: (p', p) |∈| (eps A)|^+| using p'(2) by auto
    then have eps: ((Some p', Some q'), Some p, Some q') |∈| (eps (pair-automaton
A B))|^+|
      proof (rule ftrancl-map[rotated])
        fix x y
        assume (x, y) |∈| eps A
       then show ((Some x, Some q'), Some y, Some q') |∈| eps (pair-automaton
A B)
          using q'(1)[THEN rule-statesD(1), simplified]
          apply (simp add: pair-automaton-def image-iff fSigma.rep-eq)
          by fastforce
      qed
      then show ?thesis
      proof (cases q' = q)
        case True then show ?thesis using eps
          by simp
      next
        case False
        then have (q', q) |∈| (eps B)|^+| using q'(2) by auto
       then have ((Some p, Some q'), Some p, Some q) |∈| (eps (pair-automaton
A B))|^+|
          apply (rule ftrancl-map[rotated])
          using ∗[THEN eps-trancl-statesD]
          apply (simp add: p pair-automaton-def image-iff fSigma.rep-eq)
          by fastforce
```

```
        then show ?thesis using eps
          by (meson ftrancl-trans)
      qed
    qed
    have (Some f, Some g) zip-fill ps qs → (Some p', Some q') |∈| rules ?AB
      using p'(1) q'(1) by (force simp: pair-automaton-def)
    then show ?case using GFun(1)[OF nth-mem p'(4) q'(4)] p'(1 − 3)  q'(1
− 3) eps
      using 1[OF p'(4), of - B] 2[OF q'(4)]
      by (auto simp: gpair-code nth-zip-fill less-max-iff-disj
                intro!: exI[of - zip-fill ps qs] exI[of - Some p'] exI[of - Some q'])
  qed
 }
qed

lemma from-ta-der-pair-automaton:
 (None, None) |∉| ta-der (pair-automaton A B) (term-of-gterm s)
 (Some p, None) |∈| ta-der (pair-automaton A B) (term-of-gterm s) ⟹
  ∃ t. p |∈| ta-der A (term-of-gterm t) ∧ s = map-gterm (λf. (Some f, None)) t
 (None, Some q) |∈| ta-der (pair-automaton A B) (term-of-gterm s) ⟹
  ∃ u. q |∈| ta-der B (term-of-gterm u) ∧ s = map-gterm (λf. (None, Some f)) u
 (Some p, Some q) |∈| ta-der (pair-automaton A B) (term-of-gterm s) ⟹
  ∃ t u. p |∈| ta-der A (term-of-gterm t) ∧ q |∈| ta-der B (term-of-gterm u) ∧ s =
gpair t u
proof (goal-cases nn sn ns ss)
 let ?AB = pair-automaton A B
 { fix p s A B assume (Some p, None) |∈| ta-der (pair-automaton A B) (term-of-gterm
s)
   then have ∃ t. p |∈| ta-der A (term-of-gterm t) ∧ s = map-gterm (λf. (Some
f, None)) t
   proof (induct s arbitrary: p)
     case (GFun h ss)
     obtain rs sp nq where r: h rs → (sp, nq) |∈| rules (pair-automaton A B)
          sp = Some p ∧ nq = None ∨ ((sp, nq), (Some p, None)) |∈| (eps
(pair-automaton A B))|^+| length rs = length ss
       ⋀i. i < length ss ⟹ rs ! i |∈| ta-der (pair-automaton A B) (term-of-gterm
(ss ! i))
       using GFun(2) by auto
     obtain p' where pq: sp = Some p' nq = None p' = p ∨ (p', p) |∈| (eps A)|^+|
       using r(2) pair-eps-Some-None[of sp nq p A B]
       by (cases sp) auto
     obtain f ps where h: h = lift-Some-None f rs = map lift-Some-None ps
       f ps → p' |∈| rules A
     using r(1) unfolding pq(1, 2) by (auto simp: pair-automaton-def map-ta-rule-cases)
     obtain ts where ⋀i. i < length ss ⟹
       ps ! i |∈| ta-der A (term-of-gterm (ts i)) ∧ ss ! i = map-gterm (λf. (Some
f, None)) (ts i)
     using GFun(1)[OF nth-mem, of i ps ! i for i] r(4)[unfolded h(2)] r(3)[unfolded
```

h(*2*) *length-map*]
by *auto metis*
then show *?case* **using** *h*(*3*) *pq*(*3*) *r*(*3*)[*unfolded h*(*2*) *length-map*]
by (*intro exI*[*of - GFun f* (*map ts* [*0*..<*length ss*])]) (*auto simp*: *h*(*1*) *intro*!: *nth-equalityI*)
qed
} note *1* = *this*
have *2*: ∃ *u*. *q* |∈| *ta-der B* (*term-of-gterm u*) ∧ *s* = *map-gterm* (λ*g*. (*None*, *Some g*)) *u*
if (*None*, *Some q*) |∈| *ta-der ?AB* (*term-of-gterm s*) **for** *q s*
using *1*[*OF swap-pair-automaton*, *OF that*]
by (*auto simp*: *gterm.map-comp comp-def gterm.map-ident*
dest!: *arg-cong*[*of map-gterm prod.swap - - map-gterm prod.swap*, *unfolded gterm.map-comp*])
{
case *nn*
then show *?case*
by (*intro ta-der-not-reach*) (*auto simp*: *pair-automaton-def map-ta-rule-cases*)
next
case *sn* **then show** *?case* **by** (*rule 1*)
next
case *ns* **then show** *?case* **by** (*rule 2*)
next
case *ss* **then show** *?case*
proof (*induct s arbitrary*: *p q*)
case (*GFun h ss*)
obtain *rs sp sq* **where** *r*: *h rs* → (*sp*, *sq*) |∈| *rules ?AB*
sp = *Some p* ∧ *sq* = *Some q* ∨ ((*sp*, *sq*), (*Some p*, *Some q*)) |∈| (*eps ?AB*)|^+^| *length rs* = *length ss*
⋀*i*. *i* < *length ss* ⟹ *rs* ! *i* |∈| *ta-der ?AB* (*term-of-gterm* (*ss* ! *i*))
using *GFun*(*2*) **by** *auto*
from *r*(*2*) **have** *sp* ≠ *None sq* ≠ *None* **using** *pair-eps-Some-Some2*[*of* (*sp*, *sq*) *p q*]
by *auto*
then obtain *p*′ *q*′ **where** *pq*: *sp* = *Some p*′ *sq* = *Some q*′
p′ = *p* ∨ (*p*′, *p*) |∈| (*eps A*)|^+^| *q*′ = *q* ∨ (*q*′, *q*) |∈| (*eps B*)|^+^|
using *r*(*2*) *pair-eps-Some-Some*[**where** *?r* = (*Some p*, *Some q*) **and** *?A* = *A* **and** *?B* = *B*]
using *pair-eps-Some-Some2*[*of* (*sp*, *sq*) *p q*]
by (*cases sp*; *cases sq*) *auto*
obtain *f g ps qs* **where** *h*: *h* = (*Some f*, *Some g*) *rs* = *zip-fill ps qs*
f ps → *p*′ |∈| *rules A g qs* → *q*′ |∈| *rules B*
using *r*(*1*) **unfolding** *pq*(*1*,*2*) **by** (*auto simp*: *pair-automaton-def map-ta-rule-cases*)
have ∗: ∃ *t u*. *ps* ! *i* |∈| *ta-der A* (*term-of-gterm t*) ∧ *qs* ! *i* |∈| *ta-der B* (*term-of-gterm u*) ∧ *ss* ! *i* = *gpair t u*
if *i* < *length ps i* < *length qs* **for** *i*
using *GFun*(*1*)[*OF nth-mem*, *of i ps* ! *i qs* ! *i*] *r*(*4*)[*unfolded pq*(*1*,*2*) *h*(*2*), *of i*] *that*
r(*3*)[*symmetric*] **by** (*auto simp*: *nth-zip-fill h*(*2*))

{ fix i **assume** $i <$ *length ss*
then have $\exists\, t\ u.\ (i <$ *length ps* $\longrightarrow$ *ps* ! $i\ |\in|$ *ta-der A* (*term-of-gterm t*)) $\wedge$
($i <$ *length qs* $\longrightarrow$ *qs* ! $i\ |\in|$ *ta-der B* (*term-of-gterm u*)) $\wedge$
ss ! $i =$ *gpair-impl* (*if* $i <$ *length ps then Some t else None*) (*if* $i <$ *length qs then Some u else None*)
using $*$[*of i*] *1*[*of ps* ! *i A B ss* ! *i*] *2*[*of qs* ! *i ss* ! *i*] *r*(*4*)[*of i*] *r*(*3*)[*symmetric*]
by (*cases* $i <$ *length ps*; *cases* $i <$ *length qs*)
(*auto simp add*: *h*(*2*) *nth-zip-fill less-max-iff-disj gpair-code split*: *gterm.splits*) **}**
then obtain *ts us* **where** $\bigwedge i.\ i <$ *length ss* $\Longrightarrow$
($i <$ *length ps* $\longrightarrow$ *ps* ! $i\ |\in|$ *ta-der A* (*term-of-gterm* (*ts i*))) $\wedge$
($i <$ *length qs* $\longrightarrow$ *qs* ! $i\ |\in|$ *ta-der B* (*term-of-gterm* (*us i*))) $\wedge$
ss ! $i =$ *gpair-impl* (*if* $i <$ *length ps then Some* (*ts i*) *else None*) (*if* $i <$ *length qs then Some* (*us i*) *else None*)
by *metis*
moreover then have $\bigwedge i.\ i <$ *length ps* $\Longrightarrow$ *ps* ! $i\ |\in|$ *ta-der A* (*term-of-gterm* (*ts i*))
$\bigwedge i.\ i <$ *length qs* $\Longrightarrow$ *qs* ! $i\ |\in|$ *ta-der B* (*term-of-gterm* (*us i*))
using *r*(*3*)[*unfolded h*(*2*)] **by** *auto*
ultimately show *?case* **using** *h*(*3*,*4*) *pq*(*3*,*4*) *r*(*3*)[*symmetric*]
by (*intro exI*[*of - GFun f* (*map ts* [*0*..<*length ps*])] *exI*[*of - GFun g* (*map us* [*0*..<*length qs*])])
(*auto simp*: *gpair-code h*(*1*,*2*) *less-max-iff-disj nth-zip-fill intro*!: *nth-equalityI split*: *prod.splits gterm.splits*)
qed
}
qed

lemma *diagonal-automaton*:
assumes *RR1-spec A R*
shows *RR2-spec* (*fmap-funs-reg* ($\lambda f.$ (*Some f*, *Some f*)) *A*) {(*s*, *s*) | *s*. $s \in R$}
using *assms*
by (*auto simp*: *RR2-spec-def RR1-spec-def fmap-funs-reg-def fmap-funs-gta-lang map-funs-term-some-gpair* $\mathcal{L}$*-def*)

lemma *pair-automaton*:
assumes *RR1-spec A T RR1-spec B U*
shows *RR2-spec* (*pair-automaton-reg A B*) ($T \times U$)
proof −
let *?AB* = *pair-automaton* (*ta A*) (*ta B*)
{ fix *t u* **assume** *t*: $t \in T$ **and** *u*: $u \in U$
obtain *p* **where** *p*: *p* $|\in|$ *fin A p* $|\in|$ *ta-der* (*ta A*) (*term-of-gterm t*) **using** *t assms*(*1*)
by (*auto simp*: *RR1-spec-def gta-lang-def* $\mathcal{L}$*-def gta-der-def*)
obtain *q* **where** *q*: *q* $|\in|$ *fin B q* $|\in|$ *ta-der* (*ta B*) (*term-of-gterm u*) **using** *u assms*(*2*)
by (*auto simp*: *RR1-spec-def gta-lang-def* $\mathcal{L}$*-def gta-der-def*)
have *gpair t u* $\in \mathcal{L}$ (*pair-automaton-reg A B*) **using** *p*(*1*) *q*(*1*) *to-ta-der-pair-automaton*(*3*)[*OF*

```
p(2) q(2)]
      by (auto simp: pair-automaton-reg-def L-def)
  } moreover
  { fix s assume s ∈ L (pair-automaton-reg A B)
    from this[unfolded gta-lang-def L-def]
    obtain r where r: r |∈| fin (pair-automaton-reg A B) r |∈| gta-der ?AB s
      by (auto simp: pair-automaton-reg-def)
    obtain p q where pq: r = (Some p, Some q) p |∈| fin A q |∈| fin B
      using r(1) by (auto simp: pair-automaton-reg-def)
    from from-ta-der-pair-automaton(4)[OF r(2)[unfolded pq(1) gta-der-def]]
     obtain t u where p |∈| ta-der (ta A) (term-of-gterm t) q |∈| ta-der (ta B)
(term-of-gterm u) s = gpair t u
      by (elim exE conjE) note ∗ = this
    then have t ∈ L A u ∈ L B using pq(2,3)
      by (auto simp: L-def)
    then have ∃ t u. s = gpair t u ∧ t ∈ T ∧ u ∈ U using assms
      by (auto simp: RR1-spec-def ∗(3) intro!: exI[of - t, OF exI[of - u]])
  } ultimately show ?thesis by (auto simp: RR2-spec-def)
qed

lemma pair-automaton':
  shows L (pair-automaton-reg A B) = case-prod gpair ' (L A × L B)
  using pair-automaton[of A - B] by (auto simp: RR1-spec-def RR2-spec-def)
```

5.8 Collapsing

cf. *gcollapse*.

```
fun collapse-state-list where
  collapse-state-list Qn Qs [] = [[]]
| collapse-state-list Qn Qs (q # qs) = (let rec = collapse-state-list Qn Qs qs in
    (if q |∈| Qn ∧ q |∈| Qs then map (Cons None) rec @ map (Cons (Some q)) rec
     else if q |∈| Qn then map (Cons None) rec
     else if q |∈| Qs then map (Cons (Some q)) rec
     else [[]]))

lemma collapse-state-list-inner-length:
  assumes qss = collapse-state-list Qn Qs qs
    and ∀ i < length qs. qs ! i |∈| Qn ∨ qs ! i |∈| Qs
    and i < length qss
  shows length (qss ! i) = length qs using assms
proof (induct qs arbitrary: qss i)
  case (Cons q qs)
  have ∀ i<length qs. qs ! i |∈| Qn ∨ qs ! i |∈| Qs using Cons(3) by auto
  then show ?case using Cons(1)[of collapse-state-list Qn Qs qs] Cons(2−)
    by (auto simp: Let-def nth-append)
qed auto

lemma collapse-fset-inv-constr:
  assumes ∀ i < length qs'. qs ! i |∈| Qn ∧ qs' ! i = None ∨
```

```
    qs ! i |∈| Qs ∧ qs' ! i = Some (qs ! i)
    and length qs = length qs'
  shows qs' |∈| fset-of-list (collapse-state-list Qn Qs qs) using assms
proof (induct qs arbitrary: qs')
  case (Cons q qs)
  have (tl qs') |∈| fset-of-list (collapse-state-list Qn Qs qs) using Cons(2−)
    by (intro Cons(1)[of tl qs']) (cases qs', auto)
  then show ?case using Cons(2−)
    by (cases qs') (auto simp: Let-def image-def)
qed auto

lemma collapse-fset-inv-constr2:
  assumes ∀ i < length qs. qs ! i |∈| Qn ∨ qs ! i |∈| Qs
    and qs' |∈| fset-of-list (collapse-state-list Qn Qs qs) and i < length qs'
  shows qs ! i |∈| Qn ∧ qs' ! i = None ∨ qs ! i |∈| Qs ∧ qs' ! i = Some (qs ! i)
  using assms
proof (induct qs arbitrary: qs' i)
  case (Cons a qs)
  have i ≠ 0 ⟹ qs ! (i − 1) |∈| Qn ∧ tl qs' ! (i − 1) = None ∨ qs ! (i − 1) |∈|
Qs ∧ tl qs' ! (i − 1) = Some (qs ! (i − 1))
    using Cons(2−)
    by (intro Cons(1)[of tl qs' i − 1]) (auto simp: Let-def split: if-splits)
  then show ?case using Cons(2−)
    by (cases i) (auto simp: Let-def split: if-splits)
qed auto

definition collapse-rule where
  collapse-rule A Qn Qs =
    |⋃| ((λ r. fset-of-list (map (λ qs. TA-rule (r-root r) qs (Some (r-rhs r)))
(collapse-state-list Qn Qs (r-lhs-states r)))) |`|
   ffilter (λ r. (∀ i < length (r-lhs-states r). r-lhs-states r ! i |∈| Qn ∨ r-lhs-states
r ! i |∈| Qs))
     (ffilter (λ r. r-root r ≠ None) (rules A)))

definition collapse-rule-fset where
  collapse-rule-fset A Qn Qs = (λ r. TA-rule (the (r-root r)) (map the (filter (λq.
¬ Option.is-none q) (r-lhs-states r))) (the (r-rhs r))) |`|
    collapse-rule A Qn Qs

lemma collapse-rule-set-conv:
  fset (collapse-rule-fset A Qn Qs) = {TA-rule f (map the (filter (λq. ¬ Op-
tion.is-none q) qs')) q | f qs qs' q.
     TA-rule (Some f) qs q |∈| rules A ∧ length qs = length qs' ∧
     (∀ i < length qs. qs ! i |∈| Qn ∧ qs' ! i = None ∨ qs ! i |∈| Qs ∧ (qs' ! i) =
Some (qs ! i))} (is ?Ls = ?Rs)
proof
  {fix f qs' q qs assume ass: TA-rule (Some f) qs q |∈| rules A
    length qs = length qs'
    ∀ i < length qs. qs ! i |∈| Qn ∧ qs' ! i = None ∨ qs ! i |∈| Qs ∧ (qs' ! i) =
```

Some (*qs* ! *i*)
then have *TA-rule* (*Some f*) *qs′* (*Some q*) |∈| *collapse-rule A Qn Qs*
using *collapse-fset-inv-constr*[*of qs′ qs Qn Qs*] **unfolding** *collapse-rule-def*
by (*auto simp*: *fset-of-list.rep-eq fimage-iff intro*!: *bexI*[*of - TA-rule* (*Some f*) *qs q*])
then have *TA-rule f* (*map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs′*)) *q* ∈ *?Ls*
unfolding *collapse-rule-fset-def*
by (*auto simp*: *image-iff Bex-def intro*!: *exI*[*of -TA-rule* (*Some f*) *qs′* (*Some q*)])**}**
then show *?Rs* ⊆ *?Ls* **by** *blast*
next
{fix *f qs q* **assume** *ass*: *TA-rule f qs q* ∈ *?Ls*
then obtain *ps qs′* **where** *w*: *TA-rule* (*Some f*) *ps q* |∈| *rules A*
∀ *i* < *length ps*. *ps* ! *i* |∈| *Qn* ∨ *ps* ! *i* |∈| *Qs*
qs′ |∈| *fset-of-list* (*collapse-state-list Qn Qs ps*)
qs = *map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs′*)
unfolding *collapse-rule-fset-def collapse-rule-def*
by (*auto simp*: *ffUnion.rep-eq fset-of-list.rep-eq*) (*metis ta-rule.collapse*)
then have ∀ *i* < *length qs′*. *ps* ! *i* |∈| *Qn* ∧ *qs′* ! *i* = *None* ∨ *ps* ! *i* |∈| *Qs* ∧ *qs′* ! *i* = *Some* (*ps* ! *i*)
using *collapse-fset-inv-constr2*
by *metis*
moreover have *length ps* = *length qs′*
using *collapse-state-list-inner-length*[*of - Qn Qs ps*]
using *w*(*2*, *3*) *calculation*(*1*)
by (*auto simp*: *in-fset-conv-nth*)
ultimately have *TA-rule f qs q* ∈ *?Rs*
using *w*(*1*) **unfolding** *w*(*4*)
by *auto fastforce***}**
then show *?Ls* ⊆ *?Rs*
by (*intro subsetI*) (*metis* (*no-types*, *lifting*) *ta-rule.collapse*)
qed

lemma *collapse-rule-fmember* [*simp*]:
TA-rule f qs q |∈| (*collapse-rule-fset A Qn Qs*) ⟷ (∃ *qs′ ps*.
qs = *map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs′*) ∧ *TA-rule* (*Some f*) *ps q* |∈| *rules A* ∧ *length ps* = *length qs′* ∧
(∀ *i* < *length ps*. *ps* ! *i* |∈| *Qn* ∧ *qs′* ! *i* = *None* ∨ *ps* ! *i* |∈| *Qs* ∧ (*qs′* ! *i*) = *Some* (*ps* ! *i*)))
unfolding *collapse-rule-set-conv*
by *auto*

definition *Qn A* ≡ (*let S* = (*r-rhs* |‘| *ffilter* (λ *r*. *r-root r* = *None*) (*rules A*)) *in* (*eps A*)|+| |‘‘| *S* |∪| *S*)
definition *Qs A* ≡ (*let S* = (*r-rhs* |‘| *ffilter* (λ *r*. *r-root r* ≠ *None*) (*rules A*)) *in* (*eps A*)|+| |‘‘| *S* |∪| *S*)

lemma *Qn-member-iff* [*simp*]:

$q \mid\in\mid Qn\ A \longleftrightarrow (\exists\ ps\ p.\ TA\text{-}rule\ None\ ps\ p \mid\in\mid rules\ A \wedge (p = q \vee (p,\ q) \mid\in\mid (eps\ A)\mid^{+}\mid))$ (**is** *?Ls* $\longleftrightarrow$ *?Rs*)
proof −
{assume *ass*: $q \mid\in\mid Qn\ A$ **then obtain** r **where**
$r\text{-}rhs\ r = q \vee (r\text{-}rhs\ r,\ q) \mid\in\mid (eps\ A)\mid^{+}\mid\ r \mid\in\mid rules\ A\ r\text{-}root\ r = None$
by (*force simp*: *Qn-def Image-def image-def Let-def fImage.rep-eq*)
then have *?Ls* $\Longrightarrow$ *?Rs* **by** (*cases r*) *auto***}**
moreover have *?Rs* $\Longrightarrow$ *?Ls* **by** (*force simp*: *Qn-def Image-def image-def Let-def fImage.rep-eq*)
ultimately show *?thesis* **by** *blast*
qed

lemma *Qs-member-iff* [*simp*]:
$q \mid\in\mid Qs\ A \longleftrightarrow (\exists\ f\ ps\ p.\ TA\text{-}rule\ (Some\ f)\ ps\ p \mid\in\mid rules\ A \wedge (p = q \vee (p,\ q) \mid\in\mid (eps\ A)\mid^{+}\mid))$ (**is** *?Ls* $\longleftrightarrow$ *?Rs*)
proof −
{assume *ass*: $q \mid\in\mid Qs\ A$ **then obtain** $f\ r$ **where**
$r\text{-}rhs\ r = q \vee (r\text{-}rhs\ r,\ q) \mid\in\mid (eps\ A)\mid^{+}\mid\ r \mid\in\mid rules\ A\ r\text{-}root\ r = Some\ f$
by (*force simp*: *Qs-def Image-def image-def Let-def fImage.rep-eq*)
then have *?Ls* $\Longrightarrow$ *?Rs* **by** (*cases r*) *auto***}**
moreover have *?Rs* $\Longrightarrow$ *?Ls* **by** (*force simp*: *Qs-def Image-def image-def Let-def fImage.rep-eq*)
ultimately show *?thesis* **by** *blast*
qed

lemma *collapse-Qn-Qs-set-conv*:
$fset\ (Qn\ A) = \{q' \mid qs\ q\ q'.\ TA\text{-}rule\ None\ qs\ q \mid\in\mid rules\ A \wedge (q = q' \vee (q,\ q') \mid\in\mid (eps\ A)\mid^{+}\mid)\}$ (**is** *?Ls1 = ?Rs1*)
$fset\ (Qs\ A) = \{q' \mid f\ qs\ q\ q'.\ TA\text{-}rule\ (Some\ f)\ qs\ q \mid\in\mid rules\ A \wedge (q = q' \vee (q,\ q') \mid\in\mid (eps\ A)\mid^{+}\mid)\}$ (**is** *?Ls2 = ?Rs2*)
by *auto force+*

definition *collapse-automaton* :: $('q,\ 'f\ option)\ ta \Rightarrow ('q,\ 'f)\ ta$ **where**
$collapse\text{-}automaton\ A = TA\ (collapse\text{-}rule\text{-}fset\ A\ (Qn\ A)\ (Qs\ A))\ (eps\ A)$

definition *collapse-automaton-reg* **where**
$collapse\text{-}automaton\text{-}reg\ R = Reg\ (fin\ R)\ (collapse\text{-}automaton\ (ta\ R))$

lemma *ta-states-collapse-automaton*:
$\mathcal{Q}\ (collapse\text{-}automaton\ A) \mid\subseteq\mid \mathcal{Q}\ A$
apply (*intro Q-subseteq-I*)
apply (*auto simp*: *collapse-automaton-def collapse-Qn-Qs-set-conv collapse-rule-set-conv fset-of-list.rep-eq in-set-conv-nth rule-statesD eps-statesD*[*unfolded*])
apply (*metis Option.is-none-def fnth-mem option.sel rule-statesD*(*3*) *ta-rule.sel*(*2*))
done

lemma *last-nthI*:
assumes $i < length\ ts \neg\ i < length\ ts - Suc\ 0$

```
  shows ts ! i = last ts using assms
  by (induct ts arbitrary: i)
    (auto, metis last-conv-nth length-0-conv less-antisym nth-Cons')

lemma collapse-automaton':
  assumes 𝒬 A |⊆| ta-reachable A
  shows gta-lang Q (collapse-automaton A) = the ` (gcollapse ` gta-lang Q A −
{None})
proof −
  define Qn' where Qn' = Qn A
  define Qs' where Qs' = Qs A
  {fix t assume t: t ∈ gta-lang Q (collapse-automaton A)
   then obtain q where q: q |∈| Q q |∈| ta-der (collapse-automaton A) (term-of-gterm
t) by auto
    have ∃ t'. q |∈| ta-der A (term-of-gterm t') ∧ gcollapse t' ≠ None ∧ t = the
(gcollapse t') using q(2)
    proof (induct rule: ta-der-gterm-induct)
      case (GFun f ts ps p q)
      from GFun(1 − 3) obtain qs rs where ps: TA-rule (Some f) qs p |∈| rules
A length qs = length rs
        ⋀i. i < length qs ⟹ qs ! i |∈| Qn' ∧ rs ! i = None ∨ qs ! i |∈| Qs' ∧ rs !
i = Some (qs ! i)
        ps = map the (filter (λq. ¬ Option.is-none q) rs)
        by (auto simp: collapse-automaton-def Qn'-def Qs'-def)
      obtain ts' where ts':
        ps ! i |∈| ta-der A (term-of-gterm (ts' i)) gcollapse (ts' i) ≠ None ts ! i =
the (gcollapse (ts' i))
        if i < length ts for i using GFun(5) by metis
      from ps(2, 3, 4) have rs: i < length qs ⟹ rs ! i = None ∨ rs ! i = Some
(qs ! i) for i
        by auto
      {fix i assume i < length qs rs ! i = None
        then have ∃ t'. groot-sym t' = None ∧ qs ! i |∈| ta-der A (term-of-gterm
t')
          using ps(1, 2) ps(3)[of i]
            by (auto simp: ta-der-trancl-eps Qn'-def groot-sym-groot-conv elim!:
ta-reachable-rule-gtermE[OF assms])
            (force dest: ta-der-trancl-eps)+}
      note None = this
      {fix i assume i: i < length qs rs ! i = Some (qs ! i)
        have map Some ps = filter (λq. ¬ Option.is-none q) rs using ps(4)
          by (induct rs arbitrary: ps) (auto simp add: Option.is-none-def)
        from filter-rev-nth-idx[OF - - this, of i]
        have ∗: rs ! i = map Some ps ! filter-rev-nth (λq. ¬ Option.is-none q) rs i
          filter-rev-nth (λq. ¬ Option.is-none q) rs i < length ps
          using ps(2, 4) i by auto
        then have the (rs ! i) = ps ! filter-rev-nth (λq. ¬ Option.is-none q) rs i
          filter-rev-nth (λq. ¬ Option.is-none q) rs i < length ps
          by auto} note Some = this
```

let *?P* = λ *t i*. *qs* ! *i* |∈| *ta-der A* (*term-of-gterm t*) ∧
(*rs* ! *i* = *None* ⟶ *groot-sym t* = *None*) ∧
(*rs* ! *i* = *Some* (*qs* ! *i*) ⟶ *t* = *ts'* (*filter-rev-nth* (λ*q*. ¬ *Option.is-none q*) *rs i*))
{fix *i* **assume** *i*: *i* < *length qs*
then have ∃ *t*. *?P t i* **using** *Some*[*OF i*] *None*[*OF i*] *ts' ps*(*2*, *4*) *GFun*(*2*) *rs*[*OF i*]
by (*cases rs* ! *i*) *auto***}**
then obtain *ts''* **where** *ts''*: *length ts''* = *length qs*
i < *length qs* ⟹ *qs* ! *i* |∈| *ta-der A* (*term-of-gterm* (*ts''* ! *i*))
i < *length qs* ⟹ *rs* ! *i* = *None* ⟹ *groot-sym* (*ts''* ! *i*) = *None*
i < *length qs* ⟹ *rs* ! *i* = *Some* (*qs* ! *i*) ⟹ *ts''* ! *i* = *ts'* (*filter-rev-nth* (λ*q*. ¬ *Option.is-none q*) *rs i*)
for *i* **using** *that Ex-list-of-length-P*[*of length qs ?P*] **by** *auto*
from *ts''*(*1*, *3*, *4*) *Some ps*(*2*, *4*) *GFun*(*2*) *rs ts'*(*2*−)
have *map Some ts* = (*filter* (λ*q*. ¬ *Option.is-none q*) (*map gcollapse ts''*))
proof (*induct ts''* *arbitrary*: *qs rs ps ts rule*: *rev-induct*)
case (*snoc a us*)
from *snoc*(*2*, *7*) **obtain** *r rs'* **where** [*simp*]: *rs* = *rs'* @ [*r*]
by (*metis append-butlast-last-id append-is-Nil-conv length-0-conv not-Cons-self2*)
have *l*: *length us* = *length* (*butlast qs*) *length* (*butlast qs*) = *length* (*butlast rs*)
using *snoc*(*2*, *7*) **by** *auto*
have ∗: *i* < *length* (*butlast qs*) ⟹ *butlast rs* ! *i* = *None* ⟹ *groot-sym* (*us* ! *i*) = *None* **for** *i*
using *snoc*(*3*)[*of i*] *snoc*(*2*, *7*)
by (*auto simp*:*nth-append l*(*1*) *nth-butlast split*!: *if-splits*)
have ∗∗: *i* < *length* (*butlast qs*) ⟹ *butlast rs* ! *i* = *None* ∨ *butlast rs* ! *i* = *Some* (*butlast qs* ! *i*) **for** *i*
using *snoc*(*10*)[*of i*] *snoc*(*2*, *7*) *l* **by** (*auto simp*: *nth-append nth-butlast*)
have *i* < *length* (*butlast qs*) ⟹ *butlast rs* ! *i* = *Some* (*butlast qs* ! *i*) ⟹
us ! *i* = *ts'* (*filter-rev-nth* (λ*q*. ¬ *Option.is-none q*) (*butlast rs*) *i*) **for** *i*
using *snoc*(*4*)[*of i*] *snoc*(*2*, *7*) *l*
by (*auto simp*: *nth-append nth-butlast filter-rev-nth-def take-butlast*)
note *IH* = *snoc*(*1*)[*OF l*(*1*) ∗ *this* - - *l*(*2*) - - ∗∗]
show *?case*
proof (*cases r* = *None*)
case *True*
then have *map Some ts* = *filter* (λ*q*. ¬ *Option.is-none q*) (*map gcollapse us*)
using *snoc*(*2*, *5*−)
by (*intro IH*[*of ps ts*]) (*auto simp*: *nth-append nth-butlast filter-rev-nth-butlast*)
then show *?thesis* **using** *True snoc*(*2*, *7*) *snoc*(*3*)[*of length* (*butlast rs*)]
by (*auto simp*: *nth-append l*(*1*) *last-nthI split*!: *if-splits*)
next
case *False*
then obtain *t' ss* **where** ∗: *ts* = *ss* @ [*t'*] **using** *snoc*(*2*, *7*, *8*, *9*)
by (*cases ts*) (*auto*, *metis append-butlast-last-id list.distinct*(*1*))
let *?i* = *filter-rev-nth* (λ*q*. ¬ *Option.is-none q*) *rs* (*length us*)

```
      have map Some (butlast ts) = filter (λq. ¬ Option.is-none q) (map gcollapse
us)
        using False snoc(2, 5−) l filter-rev-nth-idx
        by (intro IH[of butlast ps butlast ts])
          (fastforce simp: nth-butlast filter-rev-nth-butlast)+
       moreover have a = ts' ?i ?i < length ps
        using False snoc(2, 9) l snoc(4, 6, 10)[of length us]
        by (auto simp: nth-append)
       moreover have filter-rev-nth (λq. ¬ Option.is-none q) (rs' @ [r]) (length
rs') = length ss
        using l snoc(2, 7, 8, 9) False unfolding *
        by (auto simp: filter-rev-nth-def)
       ultimately show ?thesis using l snoc(2, 7, 9) snoc(11−)[of ?i]
        by (auto simp: nth-append *)
      qed
     qed simp
     then have ts = map the (filter (λt. ¬ Option.is-none t) (map gcollapse ts''))
      by (metis comp-the-Some list.map-id map-map)
     then show ?case using ps(1, 2) ts''(1, 2) GFun(3)
      by (auto simp: collapse-automaton-def intro!: exI[of - GFun (Some f) ts']
exI[of - qs] exI[of - p])
    qed
    then have t ∈ the ‘ (gcollapse ‘ gta-lang Q A − {None})
     by (meson Diff-iff gta-langI imageI q(1) singletonD)
  } moreover
  { fix t assume t: t ∈ gta-lang Q A gcollapse t ≠ None
    obtain q where q: q |∈| Q q |∈| ta-der A (term-of-gterm t) using t(1) by
auto
    have q |∈| ta-der (collapse-automaton A) (term-of-gterm (the (gcollapse t)))
using q(2) t(2)
    proof (induct t arbitrary: q)
     case (GFun f ts)
     obtain qs q' where q: TA-rule f qs q' |∈| rules A q' = q ∨ (q', q) |∈| (eps
(collapse-automaton A))|^+|
       length qs = length ts ⋀i. i < length ts ⟹ qs ! i |∈| ta-der A (term-of-gterm
(ts ! i))
       using GFun(2) by (auto simp: collapse-automaton-def)
      obtain f' where f [simp]: f = Some f' using GFun(3) by (cases f) auto
      define qs' where
        qs' = map (λi. if Option.is-none (gcollapse (ts ! i)) then None else Some
(qs ! i)) [0..<length qs]
      have Option.is-none (gcollapse (ts ! i)) ⟹ qs ! i |∈| Qn' if i < length qs for
i
       using q(4)[of i] that
       by (cases ts ! i rule: gcollapse.cases)
         (auto simp: q(3) Qn'-def collapse-Qn-Qs-set-conv)
      moreover have ¬ Option.is-none (gcollapse (ts ! i)) ⟹ qs ! i |∈| Qs' if i
< length qs for i
       using q(4)[of i] that
```

by (*cases ts* ! *i rule*: *gcollapse.cases*)
(*auto simp*: *q*(*3*) *Qs'-def collapse-Qn-Qs-set-conv*)
ultimately have *f'* (*map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs'*)) → *q'* |∈| *rules* (*collapse-automaton A*)
using *q*(*1*, *4*) **unfolding** *collapse-automaton-def Qn'-def*[*symmetric*] *Qs'-def*[*symmetric*]
by (*fastforce simp*: *qs'-def q*(*3*) *intro*: *exI*[*of* - *qs*] *exI*[*of* - *qs'*] *split*: *if-splits*)
moreover have ***: *length* (*filter* (λ*i*.¬ *Option.is-none* (*gcollapse* (*ts* ! *i*))) [*0*..<*length ts*]) =
length (*filter* (λ*t*. ¬ *Option.is-none* (*gcollapse t*)) *ts*) **for** *ts*
by (*subst length-map*[*of* (!) *ts*, *symmetric*] *filter-map*[*unfolded comp-def*, *symmetric*] *map-nth*)+
(*rule refl*)
note *this*[*of ts*]
moreover have *the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs'* ! *i*) |∈| *ta-der* (*collapse-automaton A*)
(*term-of-gterm* (*the* (*filter* (λ*t*. ¬ *Option.is-none t*) (*map gcollapse ts*) ! *i*)))
if *i* < *length* [*x*←*ts* . ¬ *Option.is-none* (*gcollapse x*)] **for** *i*
unfolding *qs'-def* **using** *that q*(*3*) *GFun*(*1*)[*OF nth-mem q*(*4*)]
proof (*induct ts arbitrary*: *qs rule*: *List.rev-induct*)
case (*snoc t ts*)
have *x1* [*simp*]: *i* < *length xs* ⟹ (*xs* @ *ys*) ! *i* = *xs* ! *i* **for** *xs ys i* **by** (*auto simp*: *nth-append*)
have *x2*: *i* = *length xs* ⟹ (*xs* @ *ys*) ! *i* = *ys* ! *0* **for** *xs ys i* **by** (*auto simp*: *nth-append*)
obtain *q qs'* **where** *qs* [*simp*]: *qs* = *qs'* @ [*q*] **using** *snoc*(*3*) **by** (*cases rev qs*) *auto*
have [*simp*]:
map (λ*x*. *if Option.is-none* (*gcollapse* ((*ts* @ [*t*]) ! *x*)) *then None else Some* ((*qs'* @ [*q*]) ! *x*)) [*0*..<*length ts*] =
map (λ*x*. *if Option.is-none* (*gcollapse* (*ts* ! *x*)) *then None else Some* (*qs'* ! *x*)) [*0*..<*length ts*]
using *snoc*(*3*) **by** *auto*
show *?case*
proof (*cases Option.is-none* (*gcollapse t*))
case *True* **then show** *?thesis* **using** *snoc*(*1*)[*of qs'*] *snoc*(*2*,*3*)
snoc(*4*)[*unfolded length-append list.size add-0 add-0-right add-Suc-right*, *OF less-SucI*]
by (*auto cong*: *if-cong*)
next
case *False* **note** *False'* = *this*
show *?thesis*
proof (*cases i* = *length* [*x*←*ts* . ¬ *Option.is-none* (*gcollapse x*)])
case *True*
then show *?thesis* **using** *snoc*(*3*) *snoc*(*4*)[*of length ts*]
unfolding *qs length-append list.size add-0 add-0-right add-Suc-right*
upt-Suc-append[*OF zero-le*] *filter-append map-append*
by (*subst* (*5 6*) *x2*) (*auto simp*: *comp-def* *** *False' Option.is-none-def*[*symmetric*])
next
case *False*

```
          then show ?thesis using snoc(1)[of qs'] snoc(2,3)
           snoc(4)[unfolded length-append list.size add-0 add-0-right add-Suc-right,
OF less-SucI]
            unfolding qs length-append list.size add-0 add-0-right add-Suc-right
              upt-Suc-append[OF zero-le] filter-append map-append
            by (subst (5 6) x1) (auto simp: comp-def *** False')
         qed
        qed
      qed auto
      ultimately show ?case using q(2) by (auto simp: qs'-def comp-def q(3)
        intro!: exI[of - q'] exI[of - map the (filter (λq. ¬ Option.is-none q) qs')])
    qed
    then have the (gcollapse t) ∈ gta-lang Q (collapse-automaton A)
      by (metis q(1) gta-langI)
  } ultimately show ?thesis by blast
qed

lemma L-collapse-automaton':
  assumes Q_r A |⊆| ta-reachable (ta A)
  shows L (collapse-automaton-reg A) = the ` (gcollapse ` L A − {None})
 using assms by (auto simp: collapse-automaton-reg-def L-def collapse-automaton')

lemma collapse-automaton:
  assumes Q_r A |⊆| ta-reachable (ta A) RR1-spec A T
  shows RR1-spec (collapse-automaton-reg A) (the ` (gcollapse ` L A − {None}))
  using collapse-automaton'[OF assms(1)] assms(2)
  by (simp add: collapse-automaton-reg-def L-def RR1-spec-def)
```

5.9 Cylindrification

```
definition pad-with-Nones where
  pad-with-Nones n m = (λ(f, g). case-option (replicate n None) id f @ case-option
(replicate m None) id g)

lemma gencode-append:
   gencode (ss @ ts) = map-gterm (pad-with-Nones (length ss) (length ts)) (gpair
(gencode ss) (gencode ts))
proof −
  have [simp]: p ∉ gposs (gunions (map gdomain ts)) ⟹ map (λt. gfun-at t p) ts
= replicate (length ts) None
    for p ts by (intro nth-equalityI)
      (fastforce simp: poss-gposs-mem-conv fun-at-def' image-def all-set-conv-all-nth)+
  note [simp] = glabel-map-gterm-conv[of λ(- :: unit option). (), unfolded comp-def
gdomain-id]
  show ?thesis by (auto intro!: arg-cong[of - - λx. glabel x -] simp del: gposs-gunions
    simp: pad-with-Nones-def gencode-def gunions-append gpair-def map-gterm-glabel
comp-def)
qed
```

lemma *append-automaton*:
assumes *RRn-spec n A T RRn-spec m B U*
shows *RRn-spec* (*n* + *m*) (*fmap-funs-reg* (*pad-with-Nones n m*) (*pair-automaton-reg A B*)) {*ts* @ *us* |*ts us*. *ts* ∈ *T* ∧ *us* ∈ *U*}
using *assms pair-automaton*[*of A gencode* ‘ *T B gencode* ‘ *U*]
unfolding *RRn-spec-def*
proof (*intro conjI set-eqI iffI*, *goal-cases*)
case (*1 s*)
then obtain *ts us* **where** *ts* ∈ *T us* ∈ *U s* = *gencode* (*ts* @ *us*)
by (*fastforce simp*: *L-def fmap-funs-reg-def RR1-spec-def RR2-spec-def gencode-append fmap-funs-gta-lang*)
then show *?case* **by** *blast*
qed (*fastforce simp*: *RR1-spec-def RR2-spec-def fmap-funs-reg-def L-def gencode-append fmap-funs-gta-lang*)+

lemma *cons-automaton*:
assumes *RR1-spec A T RRn-spec m B U*
shows *RRn-spec* (*Suc m*) (*fmap-funs-reg* (λ(*f*, *g*). *pad-with-Nones 1 m* (*map-option* (λ*f*. [*Some f*]) *f*, *g*))
(*pair-automaton-reg A B*)) {*t* # *us* |*t us*. *t* ∈ *T* ∧ *us* ∈ *U*}
proof −
have [*simp*]: {*ts* @ *us* |*ts us*. *ts* ∈ (λ*t*. [*t*]) ‘ *T* ∧ *us* ∈ *U*} = {*t* # *us* |*t us*. *t* ∈ *T* ∧ *us* ∈ *U*}
by (*auto intro*: *exI*[*of* - [-], *OF exI*])
show *?thesis* **using** *append-automaton*[*OF RR1-to-RRn-spec*, *OF assms*]
by (*auto simp*: *L-def fmap-funs-reg-def pair-automaton-reg-def comp-def fmap-funs-gta-lang map-pair-automaton-12 fmap-funs-ta-comp prod.case-distrib gencode-append*[*of* [-], *unfolded gencode-singleton List.append.simps*])
qed

5.10 Projection

abbreviation *drop-none-rule m fs* ≡ *if list-all* (*Option.is-none*) (*drop m fs*) *then None else Some* (*drop m fs*)

lemma *drop-automaton-reg*:
assumes $\mathcal{Q}_r$ *A* |⊆| *ta-reachable* (*ta A*) *m* < *n RRn-spec n A T*
defines *f* ≡ λ*fs*. *drop-none-rule m fs*
shows *RRn-spec* (*n* − *m*) (*collapse-automaton-reg* (*fmap-funs-reg f A*)) (*drop m* ‘ *T*)
proof −
have [*simp*]: *length ts* = *n* − *m* ==> *p* ∈ *gposs* (*gencode ts*) ⟹ ∃ *f*. ∃ *t*∈*set ts*. *Some f* = *gfun-at t p* **for** *p ts*
proof (*cases p*, *goal-cases Empty PCons*)
case *Empty*
obtain *t* **where** *t* ∈ *set ts* **using** *assms*(*2*) *Empty*(*1*) **by** (*cases ts*) *auto*
moreover then obtain *f* **where** *Some f* = *gfun-at t p* **using** *Empty*(*3*) **by** (*cases t rule*: *gterm.exhaust*) *auto*
ultimately show *?thesis* **by** *auto*

```
  next
    case (PCons i p')
    then have p ≠ [] by auto
    then show ?thesis using PCons(2)
        by (auto 0 3 simp: gencode-def eq-commute[of gfun-at - - Some -] elim!:
gfun-at-possE)
  qed
  { fix p ts y assume that: gfun-at (gencode ts) p = Some y
     have p ∈ gposs (gencode ts) ⟹ gfun-at (gencode ts) p = Some (map (λt.
gfun-at t p) ts)
      by (auto simp: gencode-def)
    moreover have gfun-at (gencode ts) p = Some y ⟹ p ∈ gposs (gencode ts)
      using gfun-at-nongposs by force
    ultimately have y = map (λt. gfun-at t p) ts ∧ p ∈ gposs (gencode ts) by
(simp add: that)
  } note [dest!] = this
  have [simp]: list-all f (replicate n x) ⟷ n = 0 ∨ f x for f n x by (induct n)
auto
  have [dest]: x ∈ set xs ⟹ list-all f xs ⟹ f x for f x xs by (induct xs) auto
  have ∗: f (pad-with-Nones m' n' z) = snd z
    if fst z = None ∨ fst z ≠ None ∧ length (the (fst z)) = m
      snd z = None ∨ snd z ≠ None ∧ length (the (snd z)) = n − m ∧ (∃ y. Some
y ∈ set (the (snd z)))
      m' = m n' = n − m for z m' n'
   using that by (auto simp: f-def pad-with-Nones-def split: option.splits prod.splits)
  { fix ts assume ts: ts ∈ T length ts = n
    have gencode (drop m ts) = the (gcollapse (map-gterm f (gencode ts)))
      gcollapse (map-gterm f (gencode ts)) ≠ None
    proof (goal-cases)
      case 1 show ?case
        using ts assms(2)
        apply (subst gsnd-gpair[of gencode (take m ts), symmetric])
      apply (subst gencode-append[of take m ts drop m ts, unfolded append-take-drop-id])
        unfolding gsnd-def comp-def gterm.map-comp
       apply (intro arg-cong[where f = λx. the (gcollapse x)] gterm.map-cong refl)
        by (subst ∗) (auto simp: gpair-def set-gterm-gposs-conv image-def)
    next
      case 2
      have [simp]: gcollapse t = None ⟷ gfun-at t [] = Some None for t
        by (cases t rule: gcollapse.cases) auto
      obtain t where t ∈ set (drop m ts) using ts(2) assms(2) by (cases drop m
ts) auto
        moreover then have ¬ Option.is-none (gfun-at t []) by (cases t rule:
gterm.exhaust) auto
      ultimately show ?case
        by (auto simp: f-def gencode-def list-all-def drop-map)
    qed
  }
  then show ?thesis using assms(3)
```

by (*fastforce simp*: $\mathcal{L}$*-def collapse-automaton-reg-def fmap-funs-reg-def RRn-spec-def fmap-funs-gta-lang gsnd-def gpair-def gterm.map-comp comp-def glabel-map-gterm-conv*[*unfolded comp-def*] *assms*(*1*) *collapse-automaton'*)
qed

lemma *gfst-collapse-simp*:
the (*gcollapse* (*map-gterm fst t*)) = *gfst t*
by (*simp add*: *gfst-def*)

lemma *gsnd-collapse-simp*:
the (*gcollapse* (*map-gterm snd t*)) = *gsnd t*
by (*simp add*: *gsnd-def*)

definition *proj-1-reg* **where**
proj-1-reg A = *collapse-automaton-reg* (*fmap-funs-reg fst* (*trim-reg A*))
definition *proj-2-reg* **where**
proj-2-reg A = *collapse-automaton-reg* (*fmap-funs-reg snd* (*trim-reg A*))

lemmas *proj-1-reg-simp* = *proj-1-reg-def collapse-automaton-reg-def fmap-funs-reg-def trim-reg-def*
lemmas *proj-2-reg-simp* = *proj-2-reg-def collapse-automaton-reg-def fmap-funs-reg-def trim-reg-def*

lemma $\mathcal{L}$*-proj-1-reg-collapse*:
$\mathcal{L}$ (*proj-1-reg* $\mathcal{A}$) = *the* ‘ (*gcollapse* ‘ *map-gterm fst* ‘ ($\mathcal{L}$ $\mathcal{A}$) − {*None*})
proof −
have $\mathcal{Q}_r$ (*fmap-funs-reg fst* (*trim-reg* $\mathcal{A}$)) |⊆| *ta-reachable* (*ta* (*fmap-funs-reg fst* (*trim-reg* $\mathcal{A}$)))
by (*auto simp*: *fmap-funs-reg-def*)
note [*simp*] = $\mathcal{L}$*-collapse-automaton'*[*OF this*]
show *?thesis* **by** (*auto simp*: *proj-1-reg-def fmap-funs-*$\mathcal{L}$ $\mathcal{L}$*-trim*)
qed

lemma $\mathcal{L}$*-proj-2-reg-collapse*:
$\mathcal{L}$ (*proj-2-reg* $\mathcal{A}$) = *the* ‘ (*gcollapse* ‘ *map-gterm snd* ‘ ($\mathcal{L}$ $\mathcal{A}$) − {*None*})
proof −
have $\mathcal{Q}_r$ (*fmap-funs-reg snd* (*trim-reg* $\mathcal{A}$)) |⊆| *ta-reachable* (*ta* (*fmap-funs-reg snd* (*trim-reg* $\mathcal{A}$)))
by (*auto simp*: *fmap-funs-reg-def*)
note [*simp*] = $\mathcal{L}$*-collapse-automaton'*[*OF this*]
show *?thesis* **by** (*auto simp*: *proj-2-reg-def fmap-funs-*$\mathcal{L}$ $\mathcal{L}$*-trim*)
qed

lemma *proj-1*:
assumes *RR2-spec A R*
shows *RR1-spec* (*proj-1-reg A*) (*fst* ‘ *R*)
proof −
{fix *s t* **assume** *ass*: (*s*, *t*) ∈ *R*
from *ass* **have** *s*: *s* = *the* (*gcollapse* (*map-gterm fst* (*gpair s t*)))

```
    by (auto simp: gfst-gpair gfst-collapse-simp)
  then have Some s = gcollapse (map-gterm fst (gpair s t))
    by (cases s; cases t) (auto simp: gpair-def)
  then have s ∈ ℒ (proj-1-reg A) using assms ass s
    by (auto simp: proj-1-reg-simp ℒ-def trim-ta-reach trim-gta-lang
      image-def image-Collect RR2-spec-def fmap-funs-gta-lang
      collapse-automaton'[of fmap-funs-ta fst (trim-ta (fin A) (ta A))])
       force}
 moreover
 {fix s assume s ∈ ℒ (proj-1-reg A) then have s ∈ fst ‘ R using assms
  by (auto simp: gfst-collapse-simp gfst-gpair rev-image-eqI RR2-spec-def trim-ta-reach
trim-gta-lang
    ℒ-def proj-1-reg-simp fmap-funs-gta-lang collapse-automaton'[of fmap-funs-ta
fst (trim-ta (fin A) (ta A))])}
 ultimately show ?thesis using assms unfolding RR2-spec-def RR1-spec-def
ℒ-def proj-1-reg-simp
  by auto
qed

lemma proj-2:
 assumes RR2-spec A R
 shows RR1-spec (proj-2-reg A) (snd ‘ R)
proof −
 {fix s t assume ass: (s, t) ∈ R
  then have s: t = the (gcollapse (map-gterm snd (gpair s t)))
    by (auto simp: gsnd-gpair gsnd-collapse-simp)
  then have Some t = gcollapse (map-gterm snd (gpair s t))
    by (cases s; cases t) (auto simp: gpair-def)
  then have t ∈ ℒ (proj-2-reg A) using assms ass s
    by (auto simp: ℒ-def trim-ta-reach trim-gta-lang proj-2-reg-simp
      image-def image-Collect RR2-spec-def fmap-funs-gta-lang
      collapse-automaton'[of fmap-funs-ta snd (trim-ta (fin A) (ta A))])
      fastforce}
 moreover
 {fix s assume s ∈ ℒ (proj-2-reg A) then have s ∈ snd ‘ R using assms
   by (auto simp: ℒ-def gsnd-collapse-simp gsnd-gpair rev-image-eqI RR2-spec-def
      trim-ta-reach trim-gta-lang proj-2-reg-simp
      fmap-funs-gta-lang collapse-automaton'[of fmap-funs-ta snd (trim-ta (fin A)
(ta A))])}
 ultimately show ?thesis using assms unfolding RR2-spec-def RR1-spec-def
  by auto
qed

lemma ℒ-proj:
 assumes RR2-spec A R
 shows ℒ (proj-1-reg A) = gfst ‘ ℒ A ℒ (proj-2-reg A) = gsnd ‘ ℒ A
proof −
 have [simp]: gfst ‘ {gpair t u |t u. (t, u) ∈ R} = fst ‘ R
  by (force simp: gfst-gpair image-def)
```

have [*simp*]: *gsnd* ‘ {*gpair t u* |*t u*. (*t*, *u*) ∈ *R*} = *snd* ‘ *R*
by (*force simp*: *gsnd-gpair image-def*)
show $\mathcal{L}$ (*proj-1-reg A*) = *gfst* ‘ $\mathcal{L}$ *A* $\mathcal{L}$ (*proj-2-reg A*) = *gsnd* ‘ $\mathcal{L}$ *A*
using *proj-1*[*OF assms*] *proj-2*[*OF assms*] *assms gfst-gpair gsnd-gpair*
by (*auto simp*: *RR1-spec-def RR2-spec-def*)
qed

lemmas *proj-automaton-gta-lang* = *proj-1 proj-2*

5.11 Permutation

lemma *gencode-permute*:
assumes *set ps* = {*0*..<*length ts*}
shows *gencode* (*map* ((!) *ts*) *ps*) = *map-gterm* (λ*xs*. *map* ((!) *xs*) *ps*) (*gencode ts*)
proof −
have ∗: (!) *ts* ‘ *set ps* = *set ts* **using** *assms* **by** (*auto simp*: *image-def set-conv-nth*)
show *?thesis* **using** *subsetD*[*OF equalityD1*[*OF assms*]]
apply (*intro eq-gterm-by-gposs-gfun-at*)
unfolding *gencode-def gposs-glabel gposs-map-gterm gposs-gunions gfun-at-map-gterm gfun-at-glabel*
set-map ∗ **by** *auto*
qed

lemma *permute-automaton*:
assumes *RRn-spec n A T set ps* = {*0*..<*n*}
shows *RRn-spec* (*length ps*) (*fmap-funs-reg* (λ*xs*. *map* ((!) *xs*) *ps*) *A*) ((λ*xs*. *map* ((!) *xs*) *ps*) ‘ *T*)
using *assms* **by** (*auto simp*: *RRn-spec-def gencode-permute fmap-funs-reg-def* $\mathcal{L}$*-def fmap-funs-gta-lang image-def*)

5.12 Intersection

lemma *intersect-automaton*:
assumes *RRn-spec n A T RRn-spec n B U*
shows *RRn-spec n* (*reg-intersect A B*) (*T* ∩ *U*) **using** *assms*
by (*simp add*: *RRn-spec-def* $\mathcal{L}$*-intersect*)
(*metis gdecode-gencode image-Int inj-on-def*)

lemma *union-automaton*:
assumes *RRn-spec n A T RRn-spec n B U*
shows *RRn-spec n* (*reg-union A B*) (*T* ∪ *U*)
using *assms* **by** (*auto simp*: *RRn-spec-def* $\mathcal{L}$*-union*)

5.13 Difference

lemma *RR1-difference*:
assumes *RR1-spec A T RR1-spec B U*

shows *RR1-spec* (*difference-reg A B*) (*T* − *U*)
using *assms* **by** (*auto simp*: *RR1-spec-def* $\mathcal{L}$*-difference-reg*)

lemma *RR2-difference*:
assumes *RR2-spec A T RR2-spec B U*
shows *RR2-spec* (*difference-reg A B*) (*T* − *U*)
using *assms* **by** (*auto simp*: *RR2-spec-def* $\mathcal{L}$*-difference-reg*)

lemma *RRn-difference*:
assumes *RRn-spec n A T RRn-spec n B U*
shows *RRn-spec n* (*difference-reg A B*) (*T* − *U*)
using *assms* **by** (*auto simp*: *RRn-spec-def* $\mathcal{L}$*-difference-reg*) (*metis gdecode-gencode*)+

5.14 All terms over a signature

definition *term-automaton* :: ('*f* × *nat*) *fset* ⇒ (*unit*, '*f*) *ta* **where**
term-automaton $\mathcal{F}$ = *TA* ((λ (*f*, *n*). *TA-rule f* (*replicate n* ()) ()) |'| $\mathcal{F}$) {||}
definition *term-reg* **where**
term-reg $\mathcal{F}$ = *Reg* {|()|} (*term-automaton* $\mathcal{F}$)

lemma *term-automaton*:
RR1-spec (*term-reg* $\mathcal{F}$) ($\mathcal{T}_G$ (*fset* $\mathcal{F}$))
unfolding *RR1-spec-def gta-lang-def term-reg-def* $\mathcal{L}$*-def*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
case (*lr t*)
then have () |∈| *ta-der* (*term-automaton* $\mathcal{F}$) (*term-of-gterm t*)
by (*auto simp*: *gta-der-def*)
then show *?case*
by (*induct t*) (*auto simp*: *term-automaton-def split*: *if-splits*)
next
case (*rl t*)
then have () |∈| *ta-der* (*term-automaton* $\mathcal{F}$) (*term-of-gterm t*)
proof (*induct t rule*: $\mathcal{T}_G$*.induct*)
case (*const a*) **then show** *?case*
by (*auto simp*: *term-automaton-def image-iff intro*: *bexI*[*of* - (*a*, *0*)])
next
case (*ind f n ss*) **then show** *?case*
by (*auto simp*: *term-automaton-def image-iff intro*: *bexI*[*of* - (*f*, *n*)])
qed
then show *?case*
by (*auto simp*: *gta-der-def*)
qed

fun *true-RRn* :: ('*f* × *nat*) *fset* ⇒ *nat* ⇒ (*nat*, '*f option list*) *reg* **where**
true-RRn $\mathcal{F}$ *0* = *Reg* {|*0*|} (*TA* {|*TA-rule* [] [] *0*|} {||})
| *true-RRn* $\mathcal{F}$ (*Suc 0*) = *relabel-reg* (*fmap-funs-reg* (λ*f*. [*Some f*]) (*term-reg* $\mathcal{F}$))
| *true-RRn* $\mathcal{F}$ (*Suc n*) = *relabel-reg*
(*trim-reg* (*fmap-funs-reg* (*pad-with-Nones 1 n*) (*pair-automaton-reg* (*true-RRn* $\mathcal{F}$ *1*) (*true-RRn* $\mathcal{F}$ *n*))))

lemma *true-RRn-spec*:
RRn-spec n (*true-RRn* $\mathcal{F}$ *n*) {*ts*. *length ts* = *n* ∧ *set ts* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)}
proof (*induct* $\mathcal{F}$ *n rule*: *true-RRn.induct*)
case (*1* $\mathcal{F}$) **show** *?case*
by (*simp cong*: *conj-cong add*: *true-RR0-spec*)
next
case (*2* $\mathcal{F}$)
moreover have {*ts*. *length ts* = *1* ∧ *set ts* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)} = (λ*t*. [*t*]) ‘ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
apply (*intro equalityI subsetI*)
subgoal for *ts* **by** (*cases ts*) *auto*
by *auto*
ultimately show *?case*
using *RR1-to-RRn-spec*[*OF term-automaton*, *of* $\mathcal{F}$] **by** *auto*
next
case (*3* $\mathcal{F}$ *n*)
have [*simp*]: {*ts* @ *us* |*ts us*. *length ts* = *n* ∧ *set ts* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) ∧ *length us* = *m* ∧
set us ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)} = {*ss*. *length ss* = *n* + *m* ∧ *set ss* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)} **for** *n m*
by (*auto 0 4 intro*!: *exI*[*of* - *take n* -, *OF exI*[*of* - *drop n* -], *of* - *xs xs* **for** *xs*]
dest!: *subsetD*[*OF set-take-subset*] *subsetD*[*OF set-drop-subset*])
show *?case* **using** *append-automaton*[*OF 3*]
by *simp*
qed

5.15 RR2 composition

abbreviation *RR2-to-RRn A* ≡ *fmap-funs-reg* (λ(*f*, *g*). [*f*, *g*]) *A*
abbreviation *RRn-to-RR2 A* ≡ *fmap-funs-reg* (λ*f*. (*f* ! *0*, *f* ! *1*)) *A*
definition *rr2-compositon* **where**
rr2-compositon $\mathcal{F}$ *A B* =
(*let A*′ = *RR2-to-RRn A in*
let B′ = *RR2-to-RRn B in*
let F = *true-RRn* $\mathcal{F}$ *1 in*
let CA = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 2 1*) (*pair-automaton-reg A*′ *F*)) *in*
let CB = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 1 2*) (*pair-automaton-reg F B*′)) *in*
let PI = *trim-reg* (*fmap-funs-reg* (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) (*reg-intersect CA CB*)) *in*
RRn-to-RR2 (*collapse-automaton-reg* (*fmap-funs-reg* (*drop-none-rule 1*) *PI*))
)

lemma *list-length1E*:
assumes *length xs* = *Suc 0* **obtains** *x* **where** *xs* = [*x*] **using** *assms*
by (*cases xs*) *auto*

lemma *rr2-compositon*:
assumes $\mathcal{R} \subseteq \mathcal{T}_G$ (*fset* $\mathcal{F}$) $\times$ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) $\mathfrak{L} \subseteq \mathcal{T}_G$ (*fset* $\mathcal{F}$) $\times$ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
and *RR2-spec A* $\mathcal{R}$ **and** *RR2-spec B* $\mathfrak{L}$
shows *RR2-spec* (*rr2-compositon* $\mathcal{F}$ *A B*) ($\mathcal{R}$ *O* $\mathfrak{L}$)
proof −
let *?R* = (λ(*t*, *u*). [*t*, *u*]) ‘ $\mathcal{R}$ **let** *?L* = (λ(*t*, *u*). [*t*, *u*]) ‘ $\mathfrak{L}$
let *?A* = *RR2-to-RRn A* **let** *?B* = *RR2-to-RRn B* **let** *?F* = *true-RRn* $\mathcal{F}$ *1*
let *?CA* = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 2 1*) (*pair-automaton-reg ?A ?F*))
let *?CB* = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 1 2*) (*pair-automaton-reg ?F ?B*))
let *?PI* = *trim-reg* (*fmap-funs-reg* (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) (*reg-intersect ?CA ?CB*))
let *?DR* = *collapse-automaton-reg* (*fmap-funs-reg* (*drop-none-rule 1*) *?PI*)
let *?Rs* = {*ts* @ *us* | *ts us*. *ts* ∈ *?R* ∧ (∃ *t*. *us* = [*t*] ∧ *t* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$))}
let *?Ls* = {*us* @ *ts* | *ts us*. *ts* ∈ *?L* ∧ (∃ *t*. *us* = [*t*] ∧ *t* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$))}
from *RR2-to-RRn-spec assms*(*3*, *4*)
have *rr2*: *RRn-spec 2 ?A ?R RRn-spec 2 ?B ?L* **by** *auto*
have ∗: {*ts*. *length ts* = *1* ∧ *set ts* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)} = {[*t*] | *t*. *t* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)}
by (*auto elim*!: *list-length1E*)
have *F*: *RRn-spec 1 ?F* {[*t*] | *t*. *t* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)} **using** *true-RRn-spec*[*of 1* $\mathcal{F}$]
unfolding ∗ **.**
have *RRn-spec 3 ?CA ?Rs RRn-spec 3 ?CB ?Ls*
using *append-automaton*[*OF rr2*(*1*) *F*] *append-automaton*[*OF F rr2*(*2*)]
by (*auto simp*: *numeral-3-eq-3*) (*smt* (*verit*) *Collect-cong*)
from *permute-automaton*[*OF intersect-automaton*[*OF this*], *of* [*1*, *0*, *2*]]
have *RRn-spec 3 ?PI* ((λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) ‘ (*?Rs* ∩ *?Ls*))
by (*auto simp*: *atLeast0-lessThan-Suc insert-commute numeral-2-eq-2 numeral-3-eq-3*)
from *drop-automaton-reg*[*OF - - this*, *of 1*]
have *sp*: *RRn-spec 2 ?DR* (*drop 1* ‘ (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) ‘ (*?Rs* ∩ *?Ls*))
by *auto*
{fix *s* **assume** *s* ∈ (λ(*t*, *u*). [*t*, *u*]) ‘ ($\mathcal{R}$ *O* $\mathfrak{L}$)
then obtain *t u v* **where** *comp*: *s* = [*t*, *u*] (*t*, *v*) ∈ $\mathcal{R}$ (*v*, *u*) ∈ $\mathfrak{L}$
by (*auto simp*: *image-iff relcomp-unfold split*!: *prod.split*)
then have [*t*, *v*] ∈ *?R* [*v* , *u*] ∈ *?L u* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) *v* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) *t* ∈ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) **using** *assms*(*1*, *2*)
by (*auto simp*: *image-iff relcomp-unfold split*!: *prod.splits*)
then have [*t*, *v*, *u*] ∈ *?Rs* [*t*, *v*, *u*] ∈ *?Ls*
apply (*simp-all*)
subgoal
apply (*rule exI*[*of* - [*t*, *v*]], *rule exI*[*of* - [*u*]])
apply *simp*
done
subgoal
apply (*rule exI*[*of* - [*v*, *u*]], *rule exI*[*of* - [*t*]])
apply *simp*
done
done
then have *s* ∈ *drop 1* ‘ (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) ‘ (*?Rs* ∩ *?Ls*) **unfolding**

comp(*1*)
apply (*simp add*: *image-def Bex-def*)
apply (*rule exI*[*of* - [*v*, *t*, *u*]]) **apply** *simp*
apply (*rule exI*[*of* - [*t*, *v*, *u*]]) **apply** *simp*
done}
moreover have *drop 1 ‘* (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) *‘* (*?Rs* ∩ *?Ls*) ⊆ (λ(*t*, *u*). [*t*, *u*]) *‘* (*R O L*)
by (*auto simp*: *image-iff relcomp-unfold Bex-def split*!: *prod.splits*)
ultimately have ∗: *drop 1 ‘* (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) *‘* (*?Rs* ∩ *?Ls*) = (λ(*t*, *u*). [*t*, *u*]) *‘* (*R O L*)
by (*simp add*: *subsetI subset-antisym*)
have ∗∗: (λ*f*. (*f* ! *0*, *f* ! *1*)) *‘* (λ(*t*, *u*). [*t*, *u*]) *‘* (*R O L*) = *R O L*
by (*force simp*: *image-def relcomp-unfold split*!: *prod.splits*)
show *?thesis* **using** *sp* **unfolding** ∗
using *RRn-to-RR2-spec*[**where** *?T* = (λ(*t*, *u*). [*t*, *u*]) *‘* (*R O L*) **and** *?A* = *?DR*]
unfolding ∗∗ **by** (*auto simp*: *rr2-compositon-def Let-def image-iff*)
qed

end
theory *RR2-Infinite*
imports *RRn-Automata Tree-Automata-Pumping*
begin

lemma *map-ta-rule-id* [*simp*]: *map-ta-rule f id r* = (*r-root r*) (*map f* (*r-lhs-states r*)) → (*f* (*r-rhs r*)) **for** *f r*
by (*simp add*: *ta-rule.expand ta-rule.map-sel*(*1* − *3*))

lemma *no-upper-bound-infinite*:
assumes ∀ (*n*::*nat*). ∃ *t* ∈ *S*. *n* < *f t*
shows *infinite S*
proof (*rule ccontr*, *simp*)
assume *finite S*
then obtain *n* **where** *n* = *Max* (*f ‘ S*) ∀ *t* ∈ *S*. *f t* ≤ *n* **by** *auto*
then show *False* **using** *assms linorder-not-le* **by** *blast*
qed

lemma *set-constr-finite*:
assumes *finite F*
shows *finite* {*h x* | *x*. *x* ∈ *F* ∧ *P x*} **using** *assms*
by (*induct*) *auto*

lemma *bounded-depth-finite*:
assumes *fin-F*: *finite* $\mathcal{F}$ **and** ⋃ (*funas-term ‘ S*) ⊆ $\mathcal{F}$
and ∀ *t* ∈ *S*. *depth t* ≤ *n* **and** ∀ *t* ∈ *S*. *ground t*
shows *finite S* **using** *assms*(*2*−)

proof (*induction n arbitrary*: *S*)
case *0*
{fix *t* **assume** *elem*: $t \in S$
from *0* **have** *depth t* = *0 ground t funas-term t* $\subseteq \mathcal{F}$ **using** *elem* **by** *auto*
then have $\exists\ f.\ (f,\ 0) \in \mathcal{F} \wedge t = Fun\ f\ []$ **by** (*cases t rule*: *depth.cases*) *auto***}**
then have $S \subseteq \{Fun\ f\ []\ |f\ .\ (f,\ 0) \in \mathcal{F}\}$ **by** (*auto simp add*: *image-iff*)
from *finite-subset*[*OF this*] **show** *?case*
using *set-constr-finite*[*OF fin-F*, *of* $\lambda\ (f,\ n).\ Fun\ f\ []\ \lambda\ x.\ snd\ x = 0$]
by *auto*
next
case (*Suc n*)
from *Suc* **obtain** S' **where**
S: $S' = \{t :: ('a,\ 'b)\ term\ .\ ground\ t \wedge funas\text{-}term\ t \subseteq \mathcal{F} \wedge depth\ t \leq n\}$ *finite* S'
by (*auto simp add*: *SUP-le-iff*)
then obtain *L F* **where** *L*: *set L* = S' *set F* = $\mathcal{F}$ **using** *fin-F* **by** (*meson finite-list*)
let *?Sn* = $\{Fun\ f\ ts \mid f\ ts.\ (f,\ length\ ts) \in \mathcal{F} \wedge set\ ts \subseteq S'\}$
let *?Ln* = *concat* (*map* $(\lambda\ (f,\ n).\ map\ (\lambda\ ts.\ Fun\ f\ ts)\ (List.n\text{-}lists\ n\ L))\ F$)
{fix *t* **assume** *elem*: $t \in S$
from *Suc* **have** *depth t* $\leq$ *Suc n ground t funas-term t* $\subseteq \mathcal{F}$ **using** *elem* **by** *auto*
then have *funas-term t* $\subseteq \mathcal{F} \wedge (\forall\ x \in set\ (args\ t).\ depth\ x \leq n) \wedge ground\ t$
by (*cases t rule*: *depth.cases*) *auto*
then have $t \in$ *?Sn* $\cup$ S'
using *S* **by** (*cases t*) *auto***}** **note** *sub* = *this*
{fix *t* **assume** *elem*: $t \in$ *?Sn*
then obtain *f ts* **where** [*simp*]: *t* = *Fun f ts* **and** *invar*: $(f,\ length\ ts) \in \mathcal{F}$ *set ts* $\subseteq S'$
by *blast*
then have *Fun f ts* $\in$ *set* (*map* $(\lambda\ ts.\ Fun\ f\ ts)$ (*List.n-lists* (*length ts*) *L*)) **using** *L*(*1*)
by (*auto simp*: *image-iff set-n-lists*)
then have *t* $\in$ *set ?Ln* **using** *invar*(*1*) *L*(*2*) **by** *auto***}**
from *this sub* **have** *sub*: *?Sn* $\subseteq$ *set ?Ln S* $\subseteq$ *?Sn* $\cup$ S' **by** *blast*+
from *finite-subset*[*OF sub*(*1*)] *finite-subset*[*OF sub*(*2*)] *finite-UnI*[*of ?Sn*, *OF - S*(*2*)]
show *?case* **by** *blast*
qed

lemma *infinite-imageD*:
infinite (*f* ‘ *S*) $\Longrightarrow$ *inj-on f S* $\Longrightarrow$ *infinite S*
by *blast*

lemma *infinite-imageD2*:
infinite (*f* ‘ *S*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *infinite S*
by *blast*

lemma *infinite-inj-image-infinite*:

assumes *infinite S* **and** *inj-on f S*
shows *infinite* (*f* ‘ *S*)
using *assms finite-image-iff* **by** *blast*

lemma *infinte-no-depth-limit*:
assumes *infinite S* **and** *finite* $\mathcal{F}$
and $\forall\, t \in S$. *funas-term t* $\subseteq \mathcal{F}$ **and** $\forall\, t \in S$. *ground t*
shows $\forall\, (n::nat).\ \exists\, t \in S.\ n <$ (*depth t*)
proof(*rule allI*, *rule ccontr*)
fix *n*::*nat*
assume ¬ ($\exists\, t \in S.$ (*depth t*) > *n*)
hence $\forall\, t \in S.$ *depth t* ≤ *n* **by** *auto*
from *bounded-depth-finite*[*OF assms*(*2*) - *this*] **show** *False* **using** *assms*
by *auto*
qed

lemma *depth-gterm-conv*:
depth (*term-of-gterm t*) = *depth* (*term-of-gterm t*)
by (*metis leD nat-neq-iff poss-gposs-conv poss-length-bounded-by-depth poss-length-depth*)

lemma *funs-term-ctxt* [*simp*]:
funs-term $C\langle s\rangle$ = *funs-ctxt C* ∪ *funs-term s*
by (*induct C*) *auto*

lemma *pigeonhole-ta-infinit-terms*:
fixes *t* ::'*f gterm* **and** $\mathcal{A}$ :: ('*q*, '*f*) *ta*
defines *t'* ≡ *term-of-gterm t* :: ('*f*, '*q*) *term*
assumes *fcard* ($\mathcal{Q}$ $\mathcal{A}$) < *depth t'* **and** *q* |∈| *gta-der* $\mathcal{A}$ *t* **and** *P* (*funas-gterm t*)
shows *infinite* {*t* . *q* |∈| *gta-der* $\mathcal{A}$ *t* ∧ *P* (*funas-gterm t*)}
proof −
from *pigeonhole-tree-automata*[*OF* - *assms*(*3*)[*unfolded gta-der-def*]] *assms*(*2*,*4*)
obtain *C C2 s v p* **where** *ctxt*: *C2* ≠ □ $C\langle s\rangle = t'$ $C2\langle v\rangle = s$ **and**
loop: *p* |∈| *ta-der* $\mathcal{A}$ *v p* |∈| *ta-der* $\mathcal{A}$ $C2\langle Var\ p\rangle$ *q* |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ p\rangle$
unfolding *assms*(*1*) **by** *auto*
let *?terms* = λ *n*. $C\langle(C2\ \widehat{}\ n)\langle v\rangle\rangle$ **let** *?inner* = λ *n*. $(C2\ \widehat{}\ n)\langle v\rangle$
have *gr*: *ground-ctxt C2 ground-ctxt C ground v*
using *arg-cong*[*OF ctxt*(*2*), *of ground*] **unfolding** *ctxt*(*3*)[*symmetric*] *assms*(*1*)
by *fastforce*+
moreover have *funas*: *funas-term* (*?terms* (*Suc n*)) = *funas-term t'* **for** *n*
unfolding *ctxt*(*2*, *3*)[*symmetric*] **using** *ctxt-comp-n-pres-funas* **by** *auto*
moreover have *der*: *q* |∈| *ta-der* $\mathcal{A}$ (*?terms* (*Suc n*)) **for** *n* **using** *loop*
by (*meson ta-der-ctxt ta-der-ctxt-n-loop*)
moreover have *n* < *depth* (*?terms* (*Suc n*)) **for** *n*
by (*meson ctxt*(*1*) *ctxt-comp-n-lower-bound depth-ctxt-less-eq less-le-trans*)
ultimately have *q* |∈| *ta-der* $\mathcal{A}$ (*?terms* (*Suc n*)) ∧ *ground* (*?terms* (*Suc n*)) ∧
P (*funas-term* (*?terms* (*Suc n*))) ∧ *n* < *depth* (*?terms* (*Suc n*)) **for** *n* **using** *assms*(*4*)
by (*auto simp*: *assms*(*1*) *funas-term-of-gterm-conv*)

then have *inf*: *infinite* {*t*. *q* |∈| *ta-der* $\mathcal{A}$ *t* ∧ *ground t* ∧ *P* (*funas-term t*)}
by (*intro no-upper-bound-infinite*[*of* - *depth*]) *blast*
have *inj*: *inj-on gterm-of-term* {*t*. *q* |∈| *ta-der* $\mathcal{A}$ *t* ∧ *ground t* ∧ *P* (*funas-term t*)}
by (*intro gterm-of-term-inj*) *simp*
show *?thesis*
by (*intro infinite-super*[*OF* - *infinite-inj-image-infinite*[*OF inf inj*]])
(*auto simp*: *image-def gta-der-def funas-gterm-gterm-of-term*)
qed

lemma *gterm-to-None-Some-funas* [*simp*]:
funas-gterm (*gterm-to-None-Some t*) ⊆ (λ (*f*, *n*). ((*None*, *Some f*), *n*)) ‘ $\mathcal{F}$ ⟷ *funas-gterm t* ⊆ $\mathcal{F}$
by (*induct t*) (*auto simp*: *funas-gterm-def*, *blast*)

lemma *funas-gterm-bot-some-decomp*:
assumes *funas-gterm s* ⊆ (λ (*f*, *n*). ((*None*, *Some f*), *n*)) ‘ $\mathcal{F}$
shows ∃ *t*. *gterm-to-None-Some t* = *s* ∧ *funas-gterm t* ⊆ $\mathcal{F}$ **using** *assms*
proof (*induct s*)
case (*GFun f ts*)
from *GFun*(*1*)[*OF nth-mem*] **obtain** *ss* **where** *l*: *length ss* = *length ts* ∧ (∀ *i*<*length ts*. *gterm-to-None-Some* (*ss* ! *i*) = *ts* ! *i*)
using *Ex-list-of-length-P*[*of length ts* λ *s i*. *gterm-to-None-Some s* = *ts* ! *i*] *GFun*(*2*−)
by (*auto simp*: *funas-gterm-def*) (*meson UN-subset-iff nth-mem*)
then have *i* < *length ss* ⟹ *funas-gterm* (*ss* ! *i*) ⊆ $\mathcal{F}$ **for** *i* **using** *GFun*(*2*)
by (*auto simp*: *UN-subset-iff*) (*smt* (*verit*) *gterm-to-None-Some-funas nth-mem subsetD*)
then show *?case* **using** *GFun*(*2*−) *l*
by (*cases f*) (*force simp*: *map-nth-eq-conv UN-subset-iff dest*!: *in-set-idx intro*!: *exI*[*of* - *GFun* (*the* (*snd f*)) *ss*])
qed

definition *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$ = {*t* . *infinite* {*u*. (*t*, *u*) ∈ $\mathcal{R}$ ∧ *funas-gterm u* ⊆ *fset* $\mathcal{F}$} ∧ *funas-gterm t* ⊆ *fset* $\mathcal{F}$}

definition *Q-infty* $\mathcal{A}$ $\mathcal{F}$ = {|*q* | *q*. *infinite* {*t* | *t*. *funas-gterm t* ⊆ *fset* $\mathcal{F}$ ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some t*))}|}

lemma *Q-infty-fmember*:
q |∈| *Q-infty* $\mathcal{A}$ $\mathcal{F}$ ⟷ *infinite* {*t* | *t*. *funas-gterm t* ⊆ *fset* $\mathcal{F}$ ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some t*))}
proof −
have {*q* | *q*. *infinite* {*t* | *t*. *funas-gterm t* ⊆ *fset* $\mathcal{F}$ ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some t*))}} ⊆ *fset* ($\mathcal{Q}$ $\mathcal{A}$)
using *not-finite-existsD* **by** *fastforce*

from *finite-subset*[*OF this*] **show** *?thesis*
by (*auto simp*: *Q-infty-def*)
qed

abbreviation *q-inf-dash-intro-rules* **where**
q-inf-dash-intro-rules Q r ≡ *if* (*r-rhs r*) |∈| *Q* ∧ *fst* (*r-root r*) = *None then* {|(*r-root r*) (*map CInl* (*r-lhs-states r*)) → *CInr* (*r-rhs r*)|} *else* {||}

abbreviation *args* :: *'a list* ⇒ *nat* ⇒ (*'a* + *'a*) *list* **where**
args ≡ λ *qs i*. *map CInl* (*take i qs*) @ *CInr* (*qs* ! *i*) # *map CInl* (*drop* (*Suc i*) *qs*)

abbreviation *q-inf-dash-closure-rules* :: (*'q*, *'f*) *ta-rule* ⇒ (*'q* + *'q*, *'f*) *ta-rule list* **where**
q-inf-dash-closure-rules r ≡ (*let* (*f*, *qs*, *q*) = (*r-root r*, *r-lhs-states r*, *r-rhs r*) *in*
(*map* (λ *i*. *f* (*args qs i*) → *CInr q*) [*0* ..< *length qs*]))

definition *Inf-automata* :: (*'q*, *'f option* × *'f option*) *ta* ⇒ *'q fset* ⇒ (*'q* + *'q*, *'f option* × *'f option*) *ta* **where**
Inf-automata A *Q* = *TA*
((|⋃| (*q-inf-dash-intro-rules Q* |‘| *rules* A)) |∪| (|⋃| ((*fset-of-list* ∘ *q-inf-dash-closure-rules*) |‘| *rules* A)) |∪|
map-ta-rule CInl id |‘| *rules* A) (*map-both Inl* |‘| *eps* A |∪| *map-both CInr* |‘| *eps* A)

definition *Inf-reg* **where**
Inf-reg A *Q* = *Reg* (*CInr* |‘| *fin* A) (*Inf-automata* (*ta* A) *Q*)

lemma *Inr-Inl-rel-comp*:
map-both CInr |‘| *S* |*O*| *map-both CInl* |‘| *S* = {||} **by** *auto*

lemmas *eps-split* = *ftrancl-Un2-separatorE*[*OF Inr-Inl-rel-comp*]

lemma *Inf-automata-eps-simp* [*simp*]:
shows (*map-both Inl* |‘| *eps* A |∪| *map-both CInr* |‘| *eps* A)|^+^| =
(*map-both CInl* |‘| *eps* A)|^+^| |∪| (*map-both CInr* |‘| *eps* A)|^+^|
proof −
{fix *x y z* **assume** (*x*, *y*) |∈| (*map-both CInl* |‘| *eps* A)|^+^|
(*y*, *z*) |∈| (*map-both CInr* |‘| *eps* A)|^+^|
then have *False*
by (*metis Inl-Inr-False eps-statesI*(*1*, *2*) *eps-states-image fimageE ftranclD ftranclD2*)**}**
then show *?thesis* **by** (*auto simp*: *Inf-automata-def eps-split*)
qed

lemma *map-both-CInl-ftrancl-conv*:
(*map-both CInl* |‘| *eps* A)|^+^| = *map-both CInl* |‘| (*eps* A)|^+^|
by (*intro ftrancl-map-both-fsubset*) (*auto simp*: *finj-CInl-CInr*)

lemma *map-both-CInr-ftrancl-conv*:

(*map-both CInr* |‘| *eps* $\mathcal{A}$)|$^+$| = *map-both CInr* |‘| (*eps* $\mathcal{A}$)|$^+$|
by (*intro ftrancl-map-both-fsubset*) (*auto simp*: *finj-CInl-CInr*)

lemmas *map-both-ftrancl-conv* = *map-both-CInl-ftrancl-conv map-both-CInr-ftrancl-conv*

lemma *Inf-automata-Inl-to-eps* [*simp*]:
(*CInl p*, *CInl q*) |∈| (*map-both CInl* |‘| *eps* $\mathcal{A}$)|$^+$| ⟷ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|
(*CInr p*, *CInr q*) |∈| (*map-both CInr* |‘| *eps* $\mathcal{A}$)|$^+$| ⟷ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$|
(*CInl q*, *CInl p*) |∈| (*map-both CInr* |‘| *eps* $\mathcal{A}$)|$^+$| ⟷ *False*
(*CInr q*, *CInr p*) |∈| (*map-both CInl* |‘| *eps* $\mathcal{A}$)|$^+$| ⟷ *False*
by (*auto simp*: *map-both-ftrancl-conv dest*: *fmap-prod-fimageI*)

lemma *Inl-eps-Inr*:
(*CInl q*, *CInl p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$| ⟷ (*CInr q*, *CInr p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$|
by (*auto simp*: *Inf-automata-def*)

lemma *Inr-rhs-eps-Inr-lhs*:
assumes (*q*, *CInr p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$|
obtains *q'* **where** *q* = *CInr q'* **using** *assms ftrancl-map-both-fsubset*[*OF finj-CInl-CInr*(*1*)]
by (*cases q*) (*auto simp*: *Inf-automata-def map-both-ftrancl-conv*)

lemma *Inl-rhs-eps-Inl-lhs*:
assumes (*q*, *CInl p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$|
obtains *q'* **where** *q* = *CInl q'* **using** *assms*
by (*cases q*) (*auto simp*: *Inf-automata-def map-both-ftrancl-conv*)

lemma *Inf-automata-eps* [*simp*]:
(*CInl q*, *CInr p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$| ⟷ *False*
(*CInr q*, *CInl p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|$^+$| ⟷ *False*
by (*auto elim*: *Inr-rhs-eps-Inr-lhs Inl-rhs-eps-Inl-lhs*)

lemma *Inl-A-res-Inf-automata*:
ta-der (*fmap-states-ta CInl* $\mathcal{A}$) *t* |⊆| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *t*
proof (*rule ta-der-mono*)
show *rules* (*fmap-states-ta CInl* $\mathcal{A}$) |⊆| *rules* (*Inf-automata* $\mathcal{A}$ *Q*)
apply (*rule fsubsetI*)
by (*auto simp*: *Inf-automata-def fmap-states-ta-def' image-iff Bex-def*)
next
show *eps* (*fmap-states-ta CInl* $\mathcal{A}$) |⊆| *eps* (*Inf-automata* $\mathcal{A}$ *Q*)
by (*rule fsubsetI*) (*simp add*: *Inf-automata-def fmap-states-ta-def'*)
qed

lemma *Inl-res-A-res-Inf-automata*:
CInl |‘| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) |⊆| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) (*term-of-gterm t*)
by (*intro fsubset-trans*[*OF ta-der-fmap-states-ta-mono*[*of CInl* $\mathcal{A}$ *t*]]) (*auto simp*: *Inl-A-res-Inf-automata*)

lemma *r-rhs-CInl-args-A-rule*:
assumes $f\ qs \to CInl\ q \mid\in\mid rules\ (Inf\text{-}automata\ \mathcal{A}\ Q)$
obtains qs' **where** $qs = map\ CInl\ qs'\ f\ qs' \to q \mid\in\mid rules\ \mathcal{A}$ **using** *assms*
by (*auto simp*: *Inf-automata-def split*!: *if-splits*)

lemma *A-rule-to-dash-closure*:
assumes $f\ qs \to q \mid\in\mid rules\ \mathcal{A}$ **and** $i < length\ qs$
shows $f\ (args\ qs\ i) \to CInr\ q \mid\in\mid rules\ (Inf\text{-}automata\ \mathcal{A}\ Q)$
using *assms* **by** (*auto simp add*: *Inf-automata-def fimage-iff fBall-def upt-fset intro*!: *fBexI*[*OF - assms*(*1*)])

lemma *Inf-automata-reach-to-dash-reach*:
assumes $CInl\ p \mid\in\mid ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInl\ q)\rangle$
shows $CInr\ p \mid\in\mid ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInr\ q)\rangle$ (**is** - $\mid\in\mid$ *ta-der ?A* -)
using *assms*
proof (*induct C arbitrary*: *p*)
case (*More f ss C ts*)
from *More*(*2*) **obtain** $qs\ q'$ **where**
rule: $f\ qs \to q' \mid\in\mid rules\ ?A\ length\ qs = Suc\ (length\ ss + length\ ts)$ **and**
eps: $q' = CInl\ p \vee (q', CInl\ p) \mid\in\mid (eps\ ?A)\mid^+\mid$ **and**
reach: $\forall\ i < Suc\ (length\ ss + length\ ts).\ qs\ !\ i \mid\in\mid ta\text{-}der\ ?A\ ((ss\ @\ C\langle Var\ (CInl\ q)\rangle\ \#\ ts)\ !\ i)$
by *auto*
from *eps* **obtain** q'' **where** [*simp*]: $q' = CInl\ q''$
by (*cases q'*) (*auto simp add*: *Inf-automata-def eps-split elim*: *ftranclE converse-ftranclE*)
from *rule* **obtain** qs' **where** *args*: $qs = map\ CInl\ qs'\ f\ qs' \to q'' \mid\in\mid rules\ \mathcal{A}$
using *r-rhs-CInl-args-A-rule* **by** (*metis* ‹$q' = CInl\ q''$›)
then have $CInl\ (qs'\ !\ length\ ss) \mid\in\mid ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInl\ q)\rangle$
using *reach*
by (*auto simp*: *all-Suc-conv nth-append-Cons*) (*metis length-map less-add-Suc1 local.rule*(*2*) *nth-append-length nth-map reach*)
from *More*(*1*)[*OF this*] *More*(*2*) **show** *?case*
using *rule args eps reach A-rule-to-dash-closure*[*OF args*(*2*), *of length ss Q*]
by (*auto simp*: *Inl-eps-Inr id-take-nth-drop all-Suc-conv*
intro!: *exI*[*of - CInr q''*] *exI*[*of - map CInl* (*take* (*length ss*) *qs'*) @ *CInr* (*qs'* ! *length ss*) # *map CInl* (*drop* (*Suc* (*length ss*)) *qs'*)])
(*auto simp*: *nth-append-Cons min-def*)
qed (*auto simp*: *Inf-automata-def*)

lemma *Inf-automata-dashI*:
assumes *run* $\mathcal{A}$ *r* (*gterm-to-None-Some t*) **and** *ex-rule-state r* $\mid\in\mid$ *Q*
shows *CInr* (*ex-rule-state r*) $\mid\in\mid$ *gta-der* (*Inf-automata* $\mathcal{A}$ *Q*) (*gterm-to-None-Some t*)
proof (*cases t*)
case [*simp*]: (*GFun f ts*)
from *run-root-rule*[*OF assms*(*1*)] *run-argsD*[*OF assms*(*1*)] **have**

rule: *TA-rule* (*None*, *Some f*) (*map ex-comp-state* (*gargs r*)) (*ex-rule-state r*) |∈| *rules* $\mathcal{A}$ *length* (*gargs r*) = *length ts* **and**
reach: ∀ *i* < *length ts*. *ex-comp-state* (*gargs r* ! *i*) |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some* (*ts* ! *i*)))
by (*auto intro*!: *run-to-comp-st-gta-der*[*unfolded gta-der-def comp-def*])
from *rule assms*(2) **have** (*None*, *Some f*) (*map* (*CInl* ∘ *ex-comp-state*) (*gargs r*)) → *CInr* (*ex-rule-state r*) |∈| *rules* (*Inf-automata* $\mathcal{A}$ *Q*)
apply (*simp add*: *Inf-automata-def image-iff bex-Un*)
apply (*rule disjI1*)
by *force*
then show *?thesis* **using** *reach rule Inl-res-A-res-Inf-automata*[*of* $\mathcal{A}$ *gterm-to-None-Some* (*ts* ! *i*) *Q* **for** *i*]
by (*auto simp*: *gta-der-def intro*!: *exI*[*of* - *CInr* (*ex-rule-state r*)] *exI*[*of* - *map* (*CInl* ∘ *ex-comp-state*) (*gargs r*)])
blast
qed

lemma *Inf-automata-dash-reach-to-reach*:
assumes *p* |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *t* (**is** - |∈| *ta-der ?A* -)
shows *remove-sum p* |∈| *ta-der* $\mathcal{A}$ (*map-vars-term remove-sum t*) **using** *assms*
proof (*induct t arbitrary*: *p*)
case (*Var x*) **then show** *?case*
by (*cases p*; *cases x*) (*auto simp*: *Inf-automata-def ftrancl-map-both map-both-ftrancl-conv*)
next
case (*Fun f ss*)
from *Fun*(2) **obtain** *qs q′* **where**
rule: *f qs* → *q′* |∈| *rules ?A length qs* = *length ss* **and**
eps: *q′* = *p* ∨ (*q′*, *p*) |∈| (*eps ?A*)|$^+$| **and**
reach: ∀ *i* < *length ss*. *qs* ! *i* |∈| *ta-der ?A* (*ss* ! *i*) **by** *auto*
from *rule* **have** *r*: *f* (*map* (*remove-sum*) *qs*) → (*remove-sum q′*) |∈| *rules* $\mathcal{A}$
by (*auto simp*: *comp-def Inf-automata-def min-def id-take-nth-drop*[*symmetric*] *upt-fset*
simp flip: *drop-map take-map split*!: *if-splits*)
moreover have *remove-sum q′* = *remove-sum p* ∨ (*remove-sum q′*, *remove-sum p*) |∈| (*eps* $\mathcal{A}$)|$^+$| **using** *eps*
by (*cases is-Inl q′*; *cases is-Inl p*) (*auto elim*!: *is-InlE is-InrE*, *auto simp*: *Inf-automata-def*)
ultimately show *?case* **using** *reach rule*(2) *Fun*(1)[*OF nth-mem*, *of i qs* ! *i* **for** *i*]
by *auto* (*metis* (*mono-tags*, *lifting*) *length-map map-nth-eq-conv*)+
qed

lemma *depth-poss-split*:
assumes *Suc* (*depth* (*term-of-gterm t*) + *n*) < *depth* (*term-of-gterm u*)
shows ∃ *p q*. *p* @ *q* ∈ *gposs u* ∧ *n* < *length q* ∧ *p* ∉ *gposs t*
proof −
from *poss-length-depth* **obtain** *p m* **where** *p*: *p* ∈ *gposs u length p* = *m depth* (*term-of-gterm u*) = *m*
using *poss-gposs-conv* **by** *blast*

then obtain m' **where** *dt*: *depth* (*term-of-gterm* t) = m' **by** *blast*
from *assms dt* $p(2, 3)$ **have** *length* (*take* (*Suc* m') p) = *Suc* m'
by (*metis Suc-leI depth-gterm-conv length-take less-add-Suc1 less-imp-le-nat less-le-trans min.absorb2*)
then have *nt*: *take* (*Suc* m') $p \notin$ *gposs* t **using** *poss-length-bounded-by-depth dt depth-gterm-conv*
by (*metis Suc-n-not-le-n gposs-to-poss*)
moreover have $n <$ *length* (*drop* (*Suc* m') p) **using** *assms depth-gterm-conv dt* $p(2-)$
by (*metis add-Suc diff-diff-left length-drop zero-less-diff*)
ultimately show *?thesis* **by** (*metis append-take-drop-id* $p(1)$)
qed

lemma *Inf-to-automata*:
assumes *RR2-spec* $\mathcal{A}$ $\mathcal{R}$ **and** $t \in$ *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
shows $\exists$ u. *gpair* t $u \in \mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) (**is** $\exists$ u. *gpair* t $u \in \mathcal{L}$ *?B*)
proof −
let *?A* = *Inf-automata* (*ta* $\mathcal{A}$) (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)
let *?t-of-g* = λ t. *term-of-gterm* t :: ('*b*, '*a*) *term*
obtain n **where** *depth-card*: *depth* (*?t-of-g* t) + *fcard* ($\mathcal{Q}$ (*ta* $\mathcal{A}$)) < n **by** *auto*
from *assms*(*1*, *2*) **have** *fin*: *infinite* {u. *gpair* t $u \in \mathcal{L}$ $\mathcal{A}$ ∧ *funas-gterm* $u \subseteq$ *fset* $\mathcal{F}$}
by (*auto simp*: *RR2-spec-def Inf-branching-terms-def*)
from *infinte-no-depth-limit*[*of ?t-of-g* ‘ {u. *gpair* t $u \in \mathcal{L}$ $\mathcal{A}$ ∧ *funas-gterm* $u \subseteq$ *fset* $\mathcal{F}$} *fset* $\mathcal{F}$] *this*
have ∀ n. ∃$t \in$ *?t-of-g* ‘ {u. *gpair* t $u \in \mathcal{L}$ $\mathcal{A}$ ∧ *funas-gterm* $u \subseteq$ *fset* $\mathcal{F}$}. $n <$ *depth* t
by (*simp add*: *infinite-inj-image-infinite*[*OF fin*] *funas-term-of-gterm-conv inj-term-of-gterm*)
from *this depth-card* **obtain** u **where** *funas*: *funas-gterm* $u \subseteq$ *fset* $\mathcal{F}$ **and**
depth: *Suc* $n <$ *depth* (*?t-of-g* u) **and** *lang*: *gpair* t $u \in \mathcal{L}$ $\mathcal{A}$ **by** *auto*
have *Suc* (*depth* (*term-of-gterm* t) + *fcard* ($\mathcal{Q}$ (*ta* $\mathcal{A}$))) < *depth* (*term-of-gterm* u)
using *depth depth-card* **by** (*metis Suc-less-eq2 depth-gterm-conv less-trans*)
from *depth-poss-split*[*OF this*] **obtain** p q **where**
pos: p @ $q \in$ *gposs* u $p \notin$ *gposs* t **and** *card*: *fcard* ($\mathcal{Q}$ (*ta* $\mathcal{A}$)) < *length* q **by** *auto*
then have *gp*: *gsubt-at* (*gpair* t u) p = *gterm-to-None-Some* (*gsubt-at* u p)
using *subst-at-gpair-nt-poss-None-Some*[*of p*] **by** *force*
from *lang* **obtain** r **where** r: *run* (*ta* $\mathcal{A}$) r (*gpair* t u) *ex-comp-state* r |∈| *fin* $\mathcal{A}$
unfolding $\mathcal{L}$-*def gta-lang-def* **by** (*fastforce dest*: *gta-der-to-run*)
from *pos* **have** *p-gtu*: $p \in$ *gposs* (*gpair* t u) **and** *pu*: $p \in$ *gposs* u
using *not-gposs-append* **by** *auto*
have *qinf*: *ex-rule-state* (*gsubt-at* r p) |∈| *Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$
using *funas-gterm-gsubt-at-subseteq*[*OF pu*] *funas card*
unfolding *Q-infty-fmember gta-der-def*[*symmetric*]
by (*intro infinite-super*[*THEN infinite-imageD2*[*OF - inj-gterm-to-None-Some*],
OF - pigeonhole-ta-infinit-terms[*of ta* $\mathcal{A}$ *gsubt-at* (*gpair* t u) p -
λ t. $t \subseteq$ (λ(f, n). ((*None*, *Some* f), n)) ‘ *fset* $\mathcal{F}$,
OF - run-to-gta-der-gsubt-at(*1*)[*OF r*(*1*) *p-gtu*]]])

(*auto simp*: *poss-length-bounded-by-depth*[*of q*] *image-iff gp less-le-trans*
pos(*1*) *poss-gposs-conv pu dest*!: *funas-gterm-bot-some-decomp*)
from *Inf-automata-dashI*[*OF run-gsubt-cl*[*OF r*(*1*) *p-gtu*, *unfolded gp*] *qinf*]
have *dashI*: *CInr* (*ex-rule-state* (*gsubt-at r p*)) |∈| *gta-der* (*Inf-automata* (*ta* $\mathcal{A}$) (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) (*gsubt-at* (*gpair t u*) *p*)
unfolding *gp*[*symmetric*] **.**
have *CInl* (*ex-comp-state r*) |∈| *ta-der ?A* (*ctxt-at-pos* (*term-of-gterm* (*gpair t u*)) *p*)⟨*Var* (*CInl* (*ex-rule-state* (*gsubt-at r p*)))⟩
using *ta-der-fmap-states-ta*[*OF run-ta-der-ctxt-split2*[*OF r*(*1*) *p-gtu*], *of CInl*, *THEN fsubsetD*[*OF Inl-A-res-Inf-automata*]]
unfolding *replace-term-at-replace-at-conv*[*OF gposs-to-poss*[*OF p-gtu*]]
by (*auto simp*: *gterm.map-ident simp flip*: *map-term-replace-at-dist*[*OF gposs-to-poss*[*OF p-gtu*]])
from *ta-der-ctxt*[*OF dashI*[*unfolded gta-der-def*] *Inf-automata-reach-to-dash-reach*[*OF this*]]
have *CInr* (*ex-comp-state r*) |∈| *gta-der* (*Inf-automata* (*ta* $\mathcal{A}$) (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) (*gpair t u*)
unfolding *replace-term-at-replace-at-conv*[*OF gposs-to-poss*[*OF p-gtu*]]
unfolding *replace-gterm-conv*[*OF p-gtu*]
by (*auto simp*: *gta-der-def*)
moreover from *r*(*2*) **have** *CInr* (*ex-comp-state r*) |∈| *fin* (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$))
by (*auto simp*: *Inf-reg-def*)
ultimately show *?thesis* **using** *r*(*2*)
by (*auto simp*: $\mathcal{L}$*-def gta-der-def Inf-reg-def intro*: *exI*[*of - u*])
qed

lemma *CInr-Inf-automata-to-q-state*:
assumes *CInr p* |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *t* **and** *ground t*
shows ∃ *C s q*. *C*⟨*s*⟩ = *t* ∧ *CInr q* |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *s* ∧ *q* |∈| *Q* ∧
CInr p |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *C*⟨*Var* (*CInr q*)⟩ ∧
(*fst* ∘ *fst* ∘ *the* ∘ *root*) *s* = *None* **using** *assms*
proof (*induct t arbitrary*: *p*)
case (*Fun f ts*)
let *?A* = (*Inf-automata* $\mathcal{A}$ *Q*)
from *Fun*(*2*) **obtain** *qs q*′ **where**
rule: *f qs* → *CInr q*′ |∈| *rules ?A length qs* = *length ts* **and**
eps: *q*′ = *p* ∨ (*CInr q*′, *CInr p*) |∈| (*eps ?A*)$^+$| **and**
reach: ∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der ?A* (*ts* ! *i*)
by *auto* (*metis Inr-rhs-eps-Inr-lhs*)
consider (*a*) ⋀ *i*. *i* < *length qs* ⟹ ∃ *q*′′. *qs* ! *i* = *CInl q*′′ | (*b*) ∃ *i* < *length qs*. ∃ *q*′′. *qs* ! *i* = *CInr q*′′
by (*meson remove-sum.cases*)
then show *?case*
proof *cases*
case *a*
then have *f qs* → *CInr q*′ |∈| |⋃| (*q-inf-dash-intro-rules Q* |‘| *rules* $\mathcal{A}$) **using** *rule*
by (*auto simp*: *Inf-automata-def min-def upt-fset split*!: *if-splits*)

(*metis* (*no-types*, *lifting*) *Inl-Inr-False Suc-pred append-eq-append-conv id-take-nth-drop*
length-Cons length-drop length-greater-0-conv length-map
less-nat-zero-code list.size(*3*) *nth-append-length rule*(*2*))
then show *?thesis* **using** *reach eps rule*
by (*intro exI*[*of - Hole*] *exI*[*of - Fun f ts*] *exI*[*of - q'*])
(*auto split*!: *if-splits*)
next
case *b*
then obtain *i q''* **where** *b*: $i < length\ ts\ qs\ !\ i = CInr\ q''$ **using** *rule*(*2*) **by** *auto*
then have $CInr\ q'' \mid\in\mid ta\text{-}der\ ?A\ (ts\ !\ i)$ **using** *rule*(*2*) *reach* **by** *auto*
from *Fun*(*3*) *Fun*(*1*)[*OF nth-mem*, *OF b*(*1*) *this*] *b rule*(*2*) **obtain** *C s q'''* **where**
ctxt: $C\langle s\rangle = ts\ !\ i$ **and**
qinf: $CInr\ q''' \mid\in\mid ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ s \wedge q''' \mid\in\mid Q$ **and**
reach2: $CInr\ q'' \mid\in\mid ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInr\ q''')\rangle$ **and**
$(fst \circ fst \circ the \circ root)\ s = None$
by *auto*
then show *?thesis* **using** *rule eps reach ctxt qinf reach2 b*(*1*) *b*(*2*)[*symmetric*]
by (*auto simp*: *min-def nth-append-Cons simp flip*: *map-append id-take-nth-drop*[*OF b*(*1*)]
intro!: *exI*[*of - More f* (*take i ts*) *C* (*drop* (*Suc i*) *ts*)] *exI*[*of - s*] *exI*[*of - q'''*] *exI*[*of - CInr q'*] *exI*[*of - qs*])
qed
qed *auto*

lemma *aux-lemma*:
assumes *RR2-spec* $\mathcal{A}\ \mathcal{R}$ **and** $\mathcal{R} \subseteq \mathcal{T}_G\ (fset\ \mathcal{F}) \times \mathcal{T}_G\ (fset\ \mathcal{F})$
and *infinite* $\{u \mid u.\ gpair\ t\ u \in \mathcal{L}\ \mathcal{A}\}$
shows $t \in Inf\text{-}branching\text{-}terms\ \mathcal{R}\ \mathcal{F}$
proof −
from *assms* **have** [*simp*]: $gpair\ t\ u \in \mathcal{L}\ \mathcal{A} \longleftrightarrow (t, u) \in \mathcal{R} \wedge u \in \mathcal{T}_G\ (fset\ \mathcal{F})$
for *u* **by** (*auto simp*: *RR2-spec-def*)
from *assms* **have** $t \in \mathcal{T}_G\ (fset\ \mathcal{F})$ **unfolding** *RR2-spec-def*
by (*auto dest*: *not-finite-existsD*)
then show *?thesis* **using** *assms* **unfolding** *Inf-branching-terms-def*
by (*auto simp*: $\mathcal{T}_G$*-equivalent-def*)
qed

lemma *Inf-automata-to-Inf*:
assumes *RR2-spec* $\mathcal{A}\ \mathcal{R}$ **and** $\mathcal{R} \subseteq \mathcal{T}_G\ (fset\ \mathcal{F}) \times \mathcal{T}_G\ (fset\ \mathcal{F})$
and $gpair\ t\ u \in \mathcal{L}\ (Inf\text{-}reg\ \mathcal{A}\ (Q\text{-}infty\ (ta\ \mathcal{A})\ \mathcal{F}))$
shows $t \in Inf\text{-}branching\text{-}terms\ \mathcal{R}\ \mathcal{F}$
proof −
let *?con* = $\lambda\ t.\ term\text{-}of\text{-}gterm\ (gterm\text{-}to\text{-}None\text{-}Some\ t)$
let *?A* = $Inf\text{-}automata\ (ta\ \mathcal{A})\ (Q\text{-}infty\ (ta\ \mathcal{A})\ \mathcal{F})$
from *assms*(*3*) **obtain** *q* **where** *fin*: $q \mid\in\mid fin\ \mathcal{A}$ **and**
reach-fin: $CInr\ q \mid\in\mid ta\text{-}der\ ?A\ (term\text{-}of\text{-}gterm\ (gpair\ t\ u))$

by (*auto simp*: *Inf-reg-def* $\mathcal{L}$*-def Inf-automata-def elim*!: *gta-langE*)
from *CInr-Inf-automata-to-q-state*[*OF reach-fin*] **obtain** $C\ s\ p$ **where**
ctxt: $C\langle s\rangle$ = *term-of-gterm* (*gpair t u*) **and**
q-inf-st: *CInr p* $|\in|$ *ta-der ?A s p* $|\in|$ *Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$ **and**
reach: *CInr q* $|\in|$ *ta-der ?A* $C\langle$*Var* (*CInr p*)$\rangle$ **and**
none: (*fst* $\circ$ *fst* $\circ$ *the* $\circ$ *root*) *s* = *None* **by** *auto*
have *gr*: *ground s ground-ctxt C* **using** *arg-cong*[*OF ctxt*, *of ground*]
by *auto*
have *reach*: *q* $|\in|$ *ta-der* (*ta* $\mathcal{A}$) (*adapt-vars-ctxt C*)$\langle$*Var p*$\rangle$
using *gr Inf-automata-dash-reach-to-reach*[*OF reach*]
by (*auto simp*: *map-vars-term-ctxt-apply*)
from *q-inf-st*(*2*) **have** *inf*: *infinite* {*v*. *funas-gterm v* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *p* $|\in|$ *ta-der* (*ta* $\mathcal{A}$) (*?con v*)}
by (*simp add*: *Q-infty-fmember*)
have *inf*: *infinite* {*v*. *funas-gterm v* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *q* $|\in|$ *gta-der* (*ta* $\mathcal{A}$) (*gctxt-of-ctxt C*)$\langle$*gterm-to-None-Some v*$\rangle_G$}
using *reach ground-ctxt-adapt-ground*[*OF gr*(*2*)] *gr*
by (*intro infinite-super*[*OF - inf*], *auto simp*: *gta-der-def*)
(*smt* (*verit*) *adapt-vars-ctxt adapt-vars-term-of-gterm ground-gctxt-of-ctxt-apply-gterm ta-der-ctxt*)
have $*$: *gfun-at* (*gterm-of-term* $C\langle s\rangle$) (*hole-pos C*) = *gfun-at* (*gterm-of-term s*) []
by (*induct C*) (*auto simp*: *nth-append-Cons*)
from *arg-cong*[*OF ctxt*, *of* λ *t*. *gfun-at* (*gterm-of-term t*) (*hole-pos C*)] *none*
have *hp-nt*: *ghole-pos* (*gctxt-of-ctxt C*) $\notin$ *gposs t* **unfolding** *ground-hole-pos-to-ghole*[*OF gr*(*2*)]
using *gfun-at-gpair*[*of t u hole-pos C*] *gr* $*$
by (*cases s*) (*auto simp flip*: *poss-gposs-mem-conv split*: *if-splits elim*: *gfun-at-possE*)
from *gpair-ctxt-decomposition*[*OF hp-nt*, *of u gsubt-at u* (*hole-pos C*)]
have *to-gpair*: *gpair t* (*gctxt-at-pos u* (*hole-pos C*))$\langle v\rangle_G$ = (*gctxt-of-ctxt C*)$\langle$*gterm-to-None-Some v*$\rangle_G$ **for** *v*
unfolding *ground-hole-pos-to-ghole*[*OF gr*(*2*)] **using** *ctxt gr*
using *subst-at-gpair-nt-poss-None-Some*[*OF - hp-nt*, *of u*]
by (*metis* (*no-types*, *lifting*) *UnE* ‹*ghole-pos* (*gctxt-of-ctxt C*) = *hole-pos C*› *gposs-of-gpair gsubt-at-gctxt-apply-ghole hole-pos-in-ctxt-apply hp-nt poss-gposs-conv term-of-gterm-ctxt-apply*)
have *inf*: *infinite* {*v*. *gpair t* ((*gctxt-at-pos u* (*hole-pos C*))$\langle v\rangle_G$) $\in$ $\mathcal{L}$ $\mathcal{A}$} **using** *fin*
by (*intro infinite-super*[*OF - inf*]) (*auto simp*: $\mathcal{L}$*-def gta-der-def simp flip*: *to-gpair*)
have *infinite* {*u* |*u*. *gpair t u* $\in$ $\mathcal{L}$ $\mathcal{A}$}
by (*intro infinite-super*[*OF - infinite-inj-image-infinite*[*OF inf gctxt-apply-inj-on-term*[*of gctxt-at-pos u* (*hole-pos C*)]]])
(*auto simp*: *image-def intro*: *infinite-super*)
then show *?thesis* **using** *assms*(*1*, *2*)
by (*intro aux-lemma*[*of* $\mathcal{A}$]) *simp*
qed

lemma *Inf-automata-subseteq*:

$\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) $\subseteq$ $\mathcal{L}$ $\mathcal{A}$ (**is** $\mathcal{L}$ *?IA* $\subseteq$ -)
proof
fix s **assume** *l*: $s \in \mathcal{L}$ *?IA*
then obtain q **where** *w*: q |∈| *fin ?IA q* |∈| *ta-der* (*ta ?IA*) (*term-of-gterm s*)
by (*auto simp*: $\mathcal{L}$*-def elim*!: *gta-langE*)
from *w*(*1*) **have** *remove-sum q* |∈| *fin* $\mathcal{A}$
by (*auto simp*: *Inf-reg-def Inf-automata-def*)
then show $s \in \mathcal{L}$ $\mathcal{A}$ **using** *Inf-automata-dash-reach-to-reach*[*of q ta* $\mathcal{A}$] *w*(*2*)
by (*auto simp*: *gterm.map-ident* $\mathcal{L}$*-def Inf-reg-def*)
(*metis gta-langI map-vars-term-term-of-gterm*)
qed

lemma $\mathcal{L}$*-Inf-reg*:
assumes *RR2-spec* $\mathcal{A}$ $\mathcal{R}$ **and** $\mathcal{R} \subseteq \mathcal{T}_G$ (*fset* $\mathcal{F}$) $\times$ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
shows *gfst* ‘ $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) = *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
proof −
{**fix** s **assume** *ass*: $s \in \mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$))
then have ∃ t u. s = *gpair t u* **using** *Inf-automata-subseteq*[*of* $\mathcal{A}$ $\mathcal{F}$] *assms*(*1*)
by (*auto simp*: *RR2-spec-def*)
then have *gfst s* ∈ *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
using *ass Inf-automata-to-Inf*[*OF assms*]
by (*force simp*: *gfst-gpair*)}
then show *?thesis* **using** *Inf-to-automata*[*OF assms*(*1*), *of* - $\mathcal{F}$]
by (*auto simp*: *gfst-gpair*) (*metis gfst-gpair image-iff*)
qed
end
theory *Tree-Automata-Abstract-Impl*
imports *Tree-Automata-Det Horn-Fset*
begin

6 Computing state derivation

lemma *ta-der-Var-code* [*code*]:
ta-der $\mathcal{A}$ (*Var q*) = *finsert q* ((*eps* $\mathcal{A}$)|$^+$| |‘| {|*q*|})
by (*auto*)

lemma *ta-der-Fun-code* [*code*]:
ta-der $\mathcal{A}$ (*Fun f ts*) =
(*let args* = *map* (*ta-der* $\mathcal{A}$) *ts in*
let P = (λ *r*. *case r of TA-rule g ps p* ⇒ *f* = *g* ∧ *list-all2 fmember ps args*) *in*
let S = *r-rhs* |‘| *ffilter P* (*rules* $\mathcal{A}$) *in*
S |∪| (*eps* $\mathcal{A}$)|$^+$| |‘| *S*) (**is** *?Ls* = *?Rs*)
proof
{**fix** q **assume** q |∈| *?Ls* **then have** q |∈| *?Rs*
apply (*simp add*: *Let-def fImage-iff fBex-def image-iff*)
by (*smt* (*verit*, *ccfv-threshold*) *IntI length-map list-all2-conv-all-nth mem-Collect-eq nth-map*
ta-rule.case ta-rule.sel(*3*))
}

then show *?Ls* |⊆| *?Rs* **by** *blast*
next
{fix *q* **assume** *q* |∈| *?Rs* **then have** *q* |∈| *?Ls*
apply (*auto simp*: *Let-def ffmember-filter fimage-iff fBex-def list-all2-conv-all-nth fImage-iff*
split!: *ta-rule.splits*)
apply (*metis ta-rule.collapse*)
apply *blast*
done}
then show *?Rs* |⊆| *?Ls* **by** *blast*
qed

definition *eps-free-automata* **where**
eps-free-automata epscl $\mathcal{A}$ =
(*let ruleps* = (λ *r*. *finsert* (*r-rhs r*) (*epscl* |‘| {|*r-rhs r*|})) *in*
let rules = (λ *r*. (λ *q*. *TA-rule* (*r-root r*) (*r-lhs-states r*) *q*) |‘| (*ruleps r*)) |‘| (*rules* $\mathcal{A}$) *in*
TA (|⋃| *rules*) {||})

lemma *eps-free* [*code*]:
eps-free $\mathcal{A}$ = *eps-free-automata* ((*eps* $\mathcal{A}$)|$^+$|) $\mathcal{A}$
apply (*intro TA-equalityI*)
apply (*auto simp*: *eps-free-def eps-free-rulep-def eps-free-automata-def*)
using *fBex-def* **apply** *fastforce*
apply (*metis ta-rule.exhaust-sel*)+
done

lemma *eps-of-eps-free-automata* [*simp*]:
eps (*eps-free-automata S* $\mathcal{A}$) = {||}
by (*auto simp add*: *eps-free-automata-def*)

lemma *eps-free-automata-empty* [*simp*]:
eps $\mathcal{A}$ = {||} ⟹ *eps-free-automata* {||} $\mathcal{A}$ = $\mathcal{A}$
by (*auto simp add*: *eps-free-automata-def intro*!: *TA-equalityI*)

7 Computing the restriction of tree automata to state set

lemma *ta-restrict* [*code*]:
ta-restrict $\mathcal{A}$ *Q* =
(*let rules* = *ffilter* (λ *r*. *case r of TA-rule f ps p* ⇒ *fset-of-list ps* |⊆| *Q* ∧ *p* |∈| *Q*) (*rules* $\mathcal{A}$) *in*
let eps = *ffilter* (λ *r*. *case r of* (*p*, *q*) ⇒ *p* |∈| *Q* ∧ *q* |∈| *Q*) (*eps* $\mathcal{A}$) *in*
TA rules eps)
by (*auto simp*: *Let-def ta-restrict-def split*!: *ta-rule.splits intro*: *finite-subset*[*OF - finite-Collect-ta-rule*])

8 Computing the epsilon transition for the product automaton

lemma *prod-eps*[*code-unfold*]:
fCollect (*prod-epsLp* $\mathcal{A}$ $\mathcal{B}$) = (λ ((*p*, *q*), *r*). ((*p*, *r*), (*q*, *r*))) |‘| (*eps* $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$)
fCollect (*prod-epsRp* $\mathcal{A}$ $\mathcal{B}$) = (λ ((*p*, *q*), *r*). ((*r*, *p*), (*r*, *q*))) |‘| (*eps* $\mathcal{B}$ |×| $\mathcal{Q}$ $\mathcal{A}$)
by (*auto simp*: *finite-prod-epsLp prod-epsLp-def finite-prod-epsRp prod-epsRp-def image-iff*
fSigma.rep-eq) *force*+

9 Computing reachability

inductive-set *ta-reach* **for** $\mathcal{A}$ **where**
rule [*intro*]: *f qs* → *q* |∈| *rules* $\mathcal{A}$ ⟹ ∀ *i* < *length qs*. *qs* ! *i* ∈ *ta-reach* $\mathcal{A}$ ⟹ *q* ∈ *ta-reach* $\mathcal{A}$
| *eps* [*intro*]: *q* ∈ *ta-reach* $\mathcal{A}$ ⟹ (*q*, *r*) |∈| *eps* $\mathcal{A}$ ⟹ *r* ∈ *ta-reach* $\mathcal{A}$

lemma *ta-reach-eps-transI*:
assumes (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| *p* ∈ *ta-reach* $\mathcal{A}$
shows *q* ∈ *ta-reach* $\mathcal{A}$ **using** *assms*
by (*induct rule*: *ftrancl-induct*) *auto*

lemma *ta-reach-ground-term-der*:
assumes *q* ∈ *ta-reach* $\mathcal{A}$
shows ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t* **using** *assms*
proof (*induct*)
case (*rule f qs q*)
then obtain *ts* **where** *length ts* = *length qs*
∀ *i* < *length qs*. *ground* (*ts* ! *i*)
∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*)
using *Ex-list-of-length-P*[*of length qs* λ *t i*. *ground t* ∧ *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *t*]
by *auto*
then show *?case* **using** *rule*(*1*)
by (*auto dest*!: *in-set-idx intro*!: *exI*[*of* - *Fun f ts*]) *blast*
qed (*auto*, *meson ta-der-eps*)

lemma *ground-term-der-ta-reach*:
assumes *ground t q* |∈| *ta-der* $\mathcal{A}$ *t*
shows *q* ∈ *ta-reach* $\mathcal{A}$ **using** *assms*(*2*, *1*)
by (*induct rule*: *ta-der-induct*) (*auto simp add*: *rule ta-reach-eps-transI*)

lemma *ta-reach-reachable*:
ta-reach $\mathcal{A}$ = *fset* (*ta-reachable* $\mathcal{A}$)
using *ta-reach-ground-term-der*[*of* - $\mathcal{A}$]
using *ground-term-der-ta-reach*[*of* - - $\mathcal{A}$]
unfolding *ta-reachable-def*
by *auto*

9.1 Horn setup for reachable states

definition *reach-rules* $\mathcal{A}$ =
  $\{qs \rightarrow_h q \mid f\ qs\ q.\ TA\text{-}rule\ f\ qs\ q\ |\in|\ rules\ \mathcal{A}\} \cup$
  $\{[q] \rightarrow_h r \mid q\ r.\ (q,\ r)\ |\in|\ eps\ \mathcal{A}\}$

locale *reach-horn* =
  fixes $\mathcal{A}$ :: $('q,\ 'f)$ *ta*
begin

sublocale *horn reach-rules* $\mathcal{A}$ **.**

lemma *reach-infer0*: *infer0* = $\{q \mid f\ q.\ TA\text{-}rule\ f\ []\ q\ |\in|\ rules\ \mathcal{A}\}$
  by (*auto simp*: *horn.infer0-def reach-rules-def*)

lemma *reach-infer1*:
  infer1 p X = $\{r \mid f\ qs\ r.\ TA\text{-}rule\ f\ qs\ r\ |\in|\ rules\ \mathcal{A} \wedge p \in set\ qs \wedge set\ qs \subseteq insert\ p\ X\} \cup$
   $\{r \mid r.\ (p,\ r)\ |\in|\ eps\ \mathcal{A}\}$
  unfolding *reach-rules-def*
  by (*auto simp*: *horn.infer1-def simp flip*: *ex-simps*(*1*))

lemma *reach-sound*:
  ta-reach $\mathcal{A}$ = *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
  case (*lr x*) **obtain** *p* **where** *x*: *p* = *ta-reach* $\mathcal{A}$ **by** *auto*
  show *?case* **using** *lr* **unfolding** *x*
  proof (*induct*)
    case (*rule f qs q*)
    then show *?case*
      by (*intro infer*[*of qs q*]) (*auto simp*: *reach-rules-def dest*: *in-set-idx*)
  next
    case (*eps q r*)
    then show *?case*
      by (*intro infer*[*of* [*q*] *r*]) (*auto simp*: *reach-rules-def*)
  qed
next
  case (*rl x*)
  then show *?case*
    by (*induct*) (*auto simp*: *reach-rules-def*)
qed
end

9.2 Computing productivity

First, use an alternative definition of productivity

inductive-set *ta-productive-ind* :: $'q$ *fset* $\Rightarrow$ $('q,'f)$ *ta* $\Rightarrow$ $'q$ *set* **for** *P* **and** $\mathcal{A}$ :: $('q,'f)$ *ta* **where**
  basic [*intro*]: *q* $|\in|$ *P* $\Longrightarrow$ *q* $\in$ *ta-productive-ind P* $\mathcal{A}$

| *eps* [*intro*]: $(p, q) \mid\in\mid (eps\ \mathcal{A})^+ \mid \Longrightarrow q \in$ *ta-productive-ind* $P\ \mathcal{A} \Longrightarrow p \in$ *ta-productive-ind* $P\ \mathcal{A}$
| *rule*: *TA-rule* $f\ qs\ q \mid\in\mid$ *rules* $\mathcal{A} \Longrightarrow q \in$ *ta-productive-ind* $P\ \mathcal{A} \Longrightarrow q' \in$ *set* qs $\Longrightarrow q' \in$ *ta-productive-ind* $P\ \mathcal{A}$

lemma *ta-productive-ind*:
ta-productive-ind $P\ \mathcal{A} =$ *fset* (*ta-productive* $P\ \mathcal{A}$) (**is** *?LS* = *?RS*)
proof −
{fix q **assume** $q \in$ *?LS* **then have** $q \in$ *?RS*
by (*induct*) (*auto dest*: *ta-prod-epsD intro*: *ta-productive-setI*,
metis (*full-types*) *in-set-conv-nth rule-reachable-ctxt-exist ta-productiveI'*)**}**
moreover
{fix q **assume** $q \in$ *?RS* **note** $* =$ *this*
from *ta-productiveE*[*OF* $*$] **obtain** $r\ C$ **where**
reach : $r \mid\in\mid$ *ta-der* $\mathcal{A}\ C\langle Var\ q\rangle$ **and** *f*: $r \mid\in\mid P$ **by** *auto*
from f **have** $r \in$ *ta-productive-ind* $P\ \mathcal{A}\ r \mid\in\mid$ *ta-productive* $P\ \mathcal{A}$
by (*auto intro*: *ta-productive-setI*)
then have $q \in$ *?LS* **using** *reach*
proof (*induct C arbitrary*: $q\ r$)
case (*More f ss C ts*)
from *iffD1 ta-der-Fun*[*THEN iffD1*, *OF More*(*4*)[*unfolded intp-actxt.simps*]]
obtain *ps p* **where**
inv: *f ps* $\rightarrow$ *p* $\mid\in\mid$ *rules* $\mathcal{A}$ *p* = *r* $\vee$ $(p, r) \mid\in\mid$ $(eps\ \mathcal{A})^+ \mid$ *length ps* = *length* (*ss* @ $C\langle Var\ q\rangle$ # *ts*)
ps ! *length ss* $\mid\in\mid$ *ta-der* $\mathcal{A}\ C\langle Var\ q\rangle$
by (*auto simp*: *nth-append-Cons split*: *if-splits*)
then have $p \in$ *ta-productive-ind* $P\ \mathcal{A} \Longrightarrow p \mid\in\mid$ *ta-der* $\mathcal{A}\ C\langle Var\ q\rangle \Longrightarrow q \in$ *ta-productive-ind* $P\ \mathcal{A}$ **for** *p*
using *More*(*1*) *calculation* **by** *auto*
note [*intro*!] = *this*[*of ps* ! *length ss*]
show *?case* **using** *More*(*2*) *inv*
by (*auto simp*: *nth-append-Cons ta-productive-ind.rule*)
(*metis less-add-Suc1 nth-mem ta-productive-ind.simps*)
qed (*auto intro*: *ta-productive-setI*)
}
ultimately show *?thesis* **by** *auto*
qed

9.2.1 Horn setup for productive states

definition *productive-rules* $P\ \mathcal{A} = \{[] \rightarrow_h q \mid q.\ q \mid\in\mid P\} \cup$
$\{[r] \rightarrow_h q \mid q\ r.\ (q, r) \mid\in\mid eps\ \mathcal{A}\} \cup$
$\{[q] \rightarrow_h r \mid f\ qs\ q\ r.$ *TA-rule* $f\ qs\ q \mid\in\mid$ *rules* $\mathcal{A} \wedge r \in$ *set* $qs\}$

locale *productive-horn* =
fixes $\mathcal{A}$ :: $('q, 'f)$ *ta* **and** P :: $'q$ *fset*
begin

sublocale *horn productive-rules* $P\ \mathcal{A}$ **.**

lemma *productive-infer0*: *infer0 = fset P*
by (*auto simp*: *productive-rules-def horn.infer0-def*)

lemma *productive-infer1*:
infer1 p X = {*r* | *r*. (*r*, *p*) |∈| *eps* $\mathcal{A}$} ∪
{*r* | *f qs r*. *TA-rule f qs p* |∈| *rules* $\mathcal{A}$ ∧ *r* ∈ *set qs*}
unfolding *productive-rules-def horn-infer1-union*
by (*auto simp add*: *horn.infer1-def*)
(*metis insertCI list.set*(*1*) *list.simps*(*15*) *singletonD subsetI*)

lemma *productive-sound*:
ta-productive-ind P $\mathcal{A}$ = *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
case (*lr p*) **then show** *?case* **using** *lr*
proof (*induct*)
case (*basic q*)
then show *?case*
by (*intro infer*[*of* [] *q*]) (*auto simp*: *productive-rules-def*)
next
case (*eps p q*) **then show** *?case*
proof (*induct rule*: *ftrancl-induct*)
case (*Base p q*)
then show *?case* **using** *infer*[*of* [*q*] *p*]
by (*auto simp*: *productive-rules-def*)
next
case (*Step p q r*)
then show *?case* **using** *infer*[*of* [*r*] *q*]
by (*auto simp*: *productive-rules-def*)
qed
next
case (*rule f qs q p*)
then show *?case*
by (*intro infer*[*of* [*q*] *p*]) (*auto simp*: *productive-rules-def*)
qed
next
case (*rl p*)
then show *?case*
by (*induct*) (*auto simp*: *productive-rules-def ta-productive-ind.rule*)
qed
end

9.3 Horn setup for power set construction states

lemma *prod-list-exists*:
assumes *fst p* ∈ *set qs set qs* ⊆ *insert* (*fst p*) (*fst* ‘ *X*)
obtains *as* **where** *p* ∈ *set as map fst as* = *qs set as* ⊆ *insert p X*
proof −
from *assms* **have** *qs* ∈ *lists* (*fst* ‘ (*insert p X*)) **by** *blast*

then obtain *ts* **where** *ts*: *map fst ts* = *qs ts* ∈ *lists* (*insert p X*)
by (*metis image-iff lists-image*)
then obtain *i* **where** *mem*: *i* < *length qs qs* ! *i* = *fst p* **using** *assms*(*1*)
by (*metis in-set-idx*)
from *ts* **have** *p*: *ts*[*i* := *p*] ∈ *lists* (*insert p X*)
using *set-update-subset-insert* **by** *fastforce*
then have *p* ∈ *set* (*ts*[*i* := *p*]) *map fst* (*ts*[*i* := *p*]) = *qs set* (*ts*[*i* := *p*]) ⊆ *insert p X*
using *mem ts*(*1*)
by (*auto simp add*: *nth-list-update set-update-memI intro*!: *nth-equalityI*)
then show *?thesis* **using** *that*
by *blast*
qed

definition *ps-states-rules* $\mathcal{A}$ = {*rs* →$_h$ (*Wrapp q*) | *rs f q*.
q = *ps-reachable-states* $\mathcal{A}$ *f* (*map ex rs*) ∧ *q* ≠ {||}}

locale *ps-states-horn* =
fixes $\mathcal{A}$ :: ('*q*, '*f*) *ta*
begin

sublocale *horn ps-states-rules* $\mathcal{A}$ **.**

lemma *ps-construction-infer0*: *infer0* =
{*Wrapp q* | *f q*. *q* = *ps-reachable-states* $\mathcal{A}$ *f* [] ∧ *q* ≠ {||}}
by (*auto simp*: *ps-states-rules-def horn.infer0-def*)

lemma *ps-construction-infer1*:
infer1 p X = {*Wrapp q* | *f qs q*. *q* = *ps-reachable-states* $\mathcal{A}$ *f* (*map ex qs*) ∧ *q* ≠ {||} ∧
p ∈ *set qs* ∧ *set qs* ⊆ *insert p X*}
unfolding *ps-states-rules-def horn-infer1-union*
by (*auto simp add*: *horn.infer1-def ps-reachable-states-def comp-def elim*!: *prod-list-exists*)

lemma *ps-states-sound*:
ps-states-set $\mathcal{A}$ = *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
case (*lr p*) **then show** *?case* **using** *lr*
proof (*induct*)
case (*1 ps f*)
then have *ps* →$_h$ (*Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*))) ∈ *ps-states-rules* $\mathcal{A}$
by (*auto simp*: *ps-states-rules-def*)
then show *?case* **using** *horn.saturate.simps 1*
by *fastforce*
qed
next
case (*rl p*)

```
  then obtain q where q ∈ saturate q = p by blast
  then show ?case
    by (induct arbitrary: p)
       (auto simp: ps-states-rules-def intro!: ps-states-set.intros)
qed

end

definition ps-reachable-states-cont where
  ps-reachable-states-cont Δ Δ_ε f ps =
   (let R = ffilter (λ r. case r of TA-rule g qs q ⇒ f = g ∧ list-all2 (|∈|) qs ps) Δ
in
    let S = r-rhs |`| R in
    S |∪| Δ_ε|^+| |`| S)

lemma ps-reachable-states [code]:
  ps-reachable-states (TA Δ Δ_ε) f ps = ps-reachable-states-cont Δ Δ_ε f ps
   by (auto simp: ps-reachable-states-fmember ps-reachable-states-cont-def Let-def
fimage-iff fBex-def
          split!: ta-rule.splits) force+

definition ps-rules-cont where
 ps-rules-cont A Q =
   (let sig = ta-sig A in
    let qss = (λ (f, n). (f, n, fset-of-list (List.n-lists n (sorted-list-of-fset Q)))) |`|
sig in
   let res = (λ (f, n, Qs). (λ qs. TA-rule f qs (Wrapp (ps-reachable-states A f (map
ex qs)))) |`| Qs) |`| qss in
      ffilter (λ r. ex (r-rhs r) ≠ {||}) ( |⋃| res))

lemma ps-rules [code]:
  ps-rules A Q = ps-rules-cont A Q
  using ps-reachable-states-sig finite-ps-rulesp-unfolded[of Q A]
  unfolding ps-rules-cont-def
  apply (auto simp: fset-of-list-elem ps-rules-def fin-mono ps-rulesp-def
    image-iff set-n-lists split!: prod.splits dest!: in-set-idx)
  by fastforce+

end
theory Tree-Automata-Class-Instances-Impl
  imports Tree-Automata
   Deriving.Compare-Instances
   Containers.Collection-Order
   Containers.Collection-Eq
   Containers.Collection-Enum
   Containers.Set-Impl
   Containers.Mapping-Impl
begin
```

derive *linorder ta-rule*
derive *linorder term*
derive *compare term*
derive (*compare*) *ccompare term*
derive *ceq ta-rule*
derive (*eq*) *ceq fset*
derive (*eq*) *ceq FSet-Lex-Wrapper*
derive (*no*) *cenum ta-rule*
derive (*no*) *cenum FSet-Lex-Wrapper*
derive *ccompare ta-rule*
derive (*eq*) *ceq term actxt*
derive (*no*) *cenum term*
derive (*rbt*) *set-impl fset FSet-Lex-Wrapper ta-rule term*

instantiation *fset* :: (*linorder*) *compare*
begin
definition *compare-fset* :: (*'a fset* ⇒ *'a fset* ⇒ *order*)
  where *compare-fset* = (λ *A B*.
    (*let A'* = *sorted-list-of-fset A in*
     let B' = *sorted-list-of-fset B in*
     if A' < *B' then Lt else if B'* < *A' then Gt else Eq*))
instance
  apply *intro-classes* **apply** (*auto simp*: *compare-fset-def comparator-def Let-def split*!: *if-splits*)
  using *sorted-list-of-fset-id* **apply** *blast+*
  done
end

instantiation *fset* :: (*linorder*) *ccompare*
begin
definition *ccompare-fset* :: (*'a fset* ⇒ *'a fset* ⇒ *order*) *option*
  where *ccompare-fset* = *Some* (λ *A B*.
    (*let A'* = *sorted-list-of-fset A in*
     let B' = *sorted-list-of-fset B in*
     if A' < *B' then Lt else if B'* < *A' then Gt else Eq*))
instance
  apply *intro-classes* **apply** (*auto simp*: *ccompare-fset-def comparator-def Let-def split*!: *if-splits*)
  using *sorted-list-of-fset-id* **apply** *blast+*
  done
end

instantiation *FSet-Lex-Wrapper* :: (*linorder*) *compare*
begin

definition *compare-FSet-Lex-Wrapper* :: *'a FSet-Lex-Wrapper* ⇒ *'a FSet-Lex-Wrapper* ⇒ *order*
  where *compare-FSet-Lex-Wrapper* = (λ *A B*.

```
    (let A' = sorted-list-of-fset (ex A) in
     let B' = sorted-list-of-fset (ex B) in
     if A' < B' then Lt else if B' < A' then Gt else Eq))

instance
  apply intro-classes apply (auto simp: compare-FSet-Lex-Wrapper-def compara-
tor-def Let-def split!: if-splits)
  using sorted-list-of-fset-id
  by (metis FSet-Lex-Wrapper.expand)
end

instantiation FSet-Lex-Wrapper :: (linorder) ccompare
begin

definition ccompare-FSet-Lex-Wrapper :: ('a FSet-Lex-Wrapper ⇒ 'a FSet-Lex-Wrapper
⇒ order) option
  where ccompare-FSet-Lex-Wrapper = Some (λ A B.
    (let A' = sorted-list-of-fset (ex A) in
     let B' = sorted-list-of-fset (ex B) in
     if A' < B' then Lt else if B' < A' then Gt else Eq))

instance
  apply intro-classes apply (auto simp: ccompare-FSet-Lex-Wrapper-def compara-
tor-def Let-def split!: if-splits)
  using sorted-list-of-fset-id
  by (metis FSet-Lex-Wrapper.expand)
end

lemma infinite-ta-rule-UNIV[simp, intro]: infinite (UNIV :: ('q,'f) ta-rule set)
proof −
  fix f :: 'f
  fix q :: 'q
  let ?map = λ n. (f (replicate n q) → q)
  have inj ?map unfolding inj-on-def by auto
  from infinite-super[OF - range-inj-infinite[OF this]]
  show ?thesis by blast
qed

instantiation ta-rule :: (type, type) card-UNIV begin
definition finite-UNIV = Phantom(('a, 'b) ta-rule) False
definition card-UNIV = Phantom(('a, 'b)ta-rule) 0
instance
  by intro-classes
   (simp-all add: infinite-ta-rule-UNIV card-UNIV-ta-rule-def finite-UNIV-ta-rule-def)
end

instantiation ta-rule :: (ccompare,ccompare)cproper-interval
begin
definition cproper-interval = (λ ( - :: ('a,'b)ta-rule option) - . False)
```

instance by (*intro-classes*, *auto*)
end

lemma *finite-finite-Fpow*:
assumes *finite A*
shows *finite* (*Fpow A*) **using** *assms*
proof (*induct A*)
case (*insert x F*)
{fix *X* **assume** *ass*: $X \subseteq$ *insert x F* $\wedge$ *finite X*
then have $X - \{x\} \subseteq F$ **using** *ass* **by** *auto*
then have *fpow* :$X - \{x\} \in$ *Fpow F* **using** *conjunct2*[*OF ass*]
by (*auto simp*: *Fpow-def*)
have $X \in$ *Fpow F* $\cup$ *insert x ' Fpow F*
proof (*cases* $x \in X$)
case *True*
then have $X \in$ *insert x ' Fpow F* **using** *fpow*
by (*metis True image-eqI insert-Diff*)
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **using** *fpow* **by** *simp*
qed}
then have $*$: *Fpow* (*insert x F*) = *Fpow F* $\cup$ *insert x ' Fpow F*
by (*auto simp add*: *Fpow-def image-def*)
show *?case* **using** *insert* **unfolding** $*$
by *simp*
qed (*auto simp*: *Fpow-def*)

lemma *infinite-infinite-Fpow*:
assumes *infinite A*
shows *infinite* (*Fpow A*)
proof −
have *inj*: *inj* (λ *S*. {*S*}) **by** *auto*
have (λ *S*. {*S*}) ' *A* $\subseteq$ *Fpow A* **by** (*auto simp*: *Fpow-def*)
from *finite-subset*[*OF this*] *inj assms*
show *?thesis*
by (*auto simp*: *finite-image-iff*)
qed

lemma *inj-on-Abs-fset*:
($\bigwedge$ *X*. $X \in A \Longrightarrow$ *finite X*) $\Longrightarrow$ *inj-on Abs-fset A* **unfolding** *inj-on-def*
by (*auto simp add*: *Abs-fset-inject*)

lemma *UNIV-FSet-Lex-Wrapper*:
(*UNIV* :: *'a FSet-Lex-Wrapper set*) = (*Wrapp* $\circ$ *Abs-fset*) ' (*Fpow* (*UNIV* :: *'a set*))
by (*simp add*: *image-def Fpow-def*) (*metis* (*mono-tags*, *lifting*) *Abs-fset-cases FSet-Lex-Wrapper.exhaust UNIV-eq-I mem-Collect-eq*)

lemma *FSet-Lex-Wrapper-UNIV*:
(*UNIV* :: *'a FSet-Lex-Wrapper set*) = (*Wrapp* ∘ *Abs-fset*) ‘ (*Fpow* (*UNIV* :: *'a set*))
by (*simp add*: *comp-def image-def Fpow-def*)
(*metis* (*mono-tags*, *lifting*) *Abs-fset-cases Abs-fset-inverse Collect-cong FSet-Lex-Wrapper.induct iso-tuple-UNIV-I mem-Collect-eq top-set-def*)

lemma *Wrapp-Abs-fset-inj*:
inj-on (*Wrapp* ∘ *Abs-fset*) (*Fpow A*)
using *inj-on-Abs-fset inj-FSet-Lex-Wrapper Fpow-def*
by (*auto simp*: *inj-on-def inj-def*)

lemma *infinite-FSet-Lex-Wrapper-UNIV*:
assumes *infinite* (*UNIV* :: *'a set*)
shows *infinite* (*UNIV* :: *'a FSet-Lex-Wrapper set*)
proof −
let *?FP* = *Fpow* (*UNIV* :: *'a set*)
have *finite* ((*Wrapp* ∘ *Abs-fset*) ‘ *?FP*) ⟹ *finite ?FP*
using *finite-image-iff*[*OF Wrapp-Abs-fset-inj*]
by (*auto simp*: *inj-on-def inj-def*)
then show *?thesis* **unfolding** *FSet-Lex-Wrapper-UNIV* **using** *infinite-infinite-Fpow*[*OF assms*]
by *auto*
qed

lemma *finite-FSet-Lex-Wrapper-UNIV*:
assumes *finite* (*UNIV* :: *'a set*)
shows *finite* (*UNIV* :: *'a FSet-Lex-Wrapper set*) **using** *assms*
unfolding *FSet-Lex-Wrapper-UNIV*
using *finite-image-iff*[*OF Wrapp-Abs-fset-inj*]
using *finite-finite-Fpow*[*OF assms*]
by *simp*

instantiation *FSet-Lex-Wrapper* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a FSet-Lex-Wrapper*)
(*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))
instance using *infinite-FSet-Lex-Wrapper-UNIV*
by *intro-classes*
(*auto simp add*: *finite-UNIV-FSet-Lex-Wrapper-def finite-UNIV finite-FSet-Lex-Wrapper-UNIV*)
end

instantiation *FSet-Lex-Wrapper* :: (*linorder*) *cproper-interval* **begin**
fun *cproper-interval-FSet-Lex-Wrapper* :: *'a FSet-Lex-Wrapper option* ⇒ *'a FSet-Lex-Wrapper option* ⇒ *bool* **where**
cproper-interval-FSet-Lex-Wrapper None None ⟷ *True*
| *cproper-interval-FSet-Lex-Wrapper None* (*Some B*) ⟷ (∃ *Z*. *sorted-list-of-fset* (*ex Z*) < *sorted-list-of-fset* (*ex B*))
| *cproper-interval-FSet-Lex-Wrapper* (*Some A*) *None* ⟷ (∃ *Z*. *sorted-list-of-fset*

(*ex A*) < *sorted-list-of-fset* (*ex Z*))
| *cproper-interval-FSet-Lex-Wrapper* (*Some A*) (*Some B*) ⟷ (∃ *Z*. *sorted-list-of-fset* (*ex A*) < *sorted-list-of-fset* (*ex Z*) ∧
sorted-list-of-fset (*ex Z*) < *sorted-list-of-fset* (*ex B*))
declare *cproper-interval-FSet-Lex-Wrapper.simps* [*code del*]

lemma *lt-of-comp-sorted-list* [*simp*]:
ID ccompare = *Some f* ⟹ *lt-of-comp f X Z* ⟷ *sorted-list-of-fset* (*ex X*) < *sorted-list-of-fset* (*ex Z*)
by (*auto simp*: *lt-of-comp-def ID-code ccompare-FSet-Lex-Wrapper-def Let-def split*!: *if-splits*)

instance by (*intro-classes*) (*auto simp*: *class.proper-interval-def*)
end

lemma *infinite-term-UNIV*[*simp*, *intro*]: *infinite* (*UNIV* :: ('*f*,'*v*)*term set*)
proof −
fix *f* :: '*f* **and** *v* :: '*v*
let *?inj* = λ*n*. *Fun f* (*replicate n* (*Var v*))
have *inj ?inj* **unfolding** *inj-on-def* **by** *auto*
from *infinite-super*[*OF - range-inj-infinite*[*OF this*]]
show *?thesis* **by** *blast*
qed

instantiation *term* :: (*type*,*type*) *finite-UNIV*
begin
definition *finite-UNIV* = *Phantom*(('*a*,'*b*)*term*) *False*
instance
by (*intro-classes*, *unfold finite-UNIV-term-def*, *simp*)
end

instantiation *term* :: (*compare*,*compare*) *cproper-interval*
begin
definition *cproper-interval* = (λ (- :: ('*a*,'*b*)*term option*) - . *False*)
instance by (*intro-classes*, *auto*)
end

derive (*assoclist*) *mapping-impl FSet-Lex-Wrapper*

end
theory *Tree-Automata-Impl*
imports *Tree-Automata-Abstract-Impl*
HOL−Library.List-Lexorder
HOL−Library.AList-Mapping
Tree-Automata-Class-Instances-Impl

Containers.Containers
begin

definition *map-val-of-list* :: ('*b* ⇒ '*a*) ⇒ ('*b* ⇒ '*c list*) ⇒ '*b list* ⇒ ('*a*, '*c list*) *mapping* **where**
map-val-of-list ek ev xs = *foldr* (λ*x m*. *Mapping.update* (*ek x*) (*ev x* @ *case-option Nil id* (*Mapping.lookup m* (*ek x*))) *m*) *xs Mapping.empty*

abbreviation *map-of-list ek ev xs* ≡ *map-val-of-list ek* (λ *x*. [*ev x*]) *xs*

lemma *map-val-of-list-tabulate-conv*:
map-val-of-list ek ev xs = *Mapping.tabulate* (*sort* (*remdups* (*map ek xs*))) (λ *k*. *concat* (*map ev* (*filter* (λ *x*. *k* = *ek x*) *xs*)))
unfolding *map-val-of-list-def*
proof (*induct xs*)
case (*Cons x xs*) **then show** *?case*
by (*intro mapping-eqI*) (*auto simp*: *lookup-combine lookup-update*′ *lookup-empty lookup-tabulate image-iff*)
qed (*simp add*: *empty-Mapping tabulate-Mapping*)

lemmas *map-val-of-list-simp* = *map-val-of-list-tabulate-conv lookup-tabulate*

9.4 Setup for the list implementation of reachable states

definition *reach-infer0-cont* **where**
reach-infer0-cont Δ =
map r-rhs (*filter* (λ *r*. *case r of TA-rule f ps p* ⇒ *ps* = []) (*sorted-list-of-fset* Δ))

definition *reach-infer1-cont* :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset* ⇒ '*q* ⇒ '*q fset* ⇒ '*q list* **where**
reach-infer1-cont Δ Δ$_\varepsilon$ =
(*let rules* = *sorted-list-of-fset* Δ *in*
let eps = *sorted-list-of-fset* Δ$_\varepsilon$ *in*
let mapp-r = *map-val-of-list fst snd* (*concat* (*map* (λ *r*. *map* (λ *q*. (*q*, [*r*])) (*r-lhs-states r*)) *rules*)) *in*
let mapp-e = *map-of-list fst snd eps in*
(λ *p bs*.
(*map r-rhs* (*filter* (λ *r*. *case r of TA-rule f qs q* ⇒
fset-of-list qs |⊆| *finsert p bs*) (*case-option Nil id* (*Mapping.lookup mapp-r p*)))) @
case-option Nil id (*Mapping.lookup mapp-e p*)))

locale *reach-rules-fset* =
fixes Δ :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* **and** Δ$_\varepsilon$ :: ('*q* × '*q*) *fset*
begin

sublocale *reach-horn TA* Δ Δ$_\varepsilon$ **.**

lemma *infer1*:
infer1 p (*fset bs*) = *set* (*reach-infer1-cont* Δ Δ_ε *p bs*)
unfolding *reach-infer1 reach-infer1-cont-def set-append Un-assoc*[*symmetric*] *Let-def*
unfolding *sorted-list-of-fset-simps union-fset*
apply (*intro arg-cong2*[*of* - - - - (∪)])
subgoal
apply (*auto simp*: *fset-of-list-elem less-eq-fset.rep-eq fset-of-list.rep-eq image-iff map-val-of-list-simp split*!: *ta-rule.splits*)
apply (*metis list.set-intros*(*1*) *ta-rule.sel*(*2*, *3*))
apply (*metis in-set-simps*(*2*) *ta-rule.exhaust-sel*)
done
subgoal
apply (*simp add*: *image-def Bex-def map-val-of-list-simp*)
done
done

sublocale *l*: *horn-fset reach-rules* (*TA* Δ Δ_ε) *reach-infer0-cont* Δ *reach-infer1-cont* Δ Δ_ε
apply (*unfold-locales*)
unfolding *reach-infer0 reach-infer0-cont-def*
subgoal
apply (*auto simp*: *image-iff ta-rule.case-eq-if Bex-def fset-of-list-elem*)
apply *force*
apply (*metis ta-rule.collapse*)+
done
subgoal using *infer1*
apply *blast*
done
done

lemmas *infer* = *l.infer0 l.infer1*
lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition *reach-cont-impl* Δ Δ_ε =
horn-fset-impl.saturate-impl (*reach-infer0-cont* Δ) (*reach-infer1-cont* Δ Δ_ε)

lemma *reach-fset-impl-sound*:
reach-cont-impl Δ Δ_ε = *Some xs* ⟹ *fset xs* = *ta-reach* (*TA* Δ Δ_ε)
using *reach-rules-fset.saturate-impl-sound* **unfolding** *reach-cont-impl-def*
unfolding *reach-horn.reach-sound* **.**

lemma *reach-fset-impl-complete*:
reach-cont-impl Δ Δ_ε ≠ *None*
proof −
have *finite* (*ta-reach* (*TA* Δ Δ_ε))
unfolding *ta-reach-reachable* **by** *simp*

then show *?thesis* **unfolding** *reach-cont-impl-def*
by (*intro reach-rules-fset.saturate-impl-complete*)
(*auto simp*: *reach-horn.reach-sound*)
qed

lemma *reach-impl* [*code*]:
ta-reachable (*TA* Δ Δ_ε) = *the* (*reach-cont-impl* Δ Δ_ε)
using *reach-fset-impl-sound*[*of* Δ Δ_ε]
by (*auto simp add*: *ta-reach-reachable reach-fset-impl-complete fset-of-list-elem*)

9.5 Setup for list implementation of productive states

definition *productive-infer1-cont* :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset* ⇒ '*q* ⇒ '*q fset* ⇒ '*q list* **where**
productive-infer1-cont Δ Δ_ε =
(*let rules* = *sorted-list-of-fset* Δ *in*
let eps = *sorted-list-of-fset* Δ_ε *in*
let mapp-r = *map-of-list* (λ *r*. *r-rhs r*) *r-lhs-states rules in*
let mapp-e = *map-of-list snd fst eps in*
(λ *p bs*.
(*case-option Nil id* (*Mapping.lookup mapp-e p*)) @
concat (*case-option Nil id* (*Mapping.lookup mapp-r p*))))

locale *productive-rules-fset* =
fixes Δ :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* **and** Δ_ε :: ('*q* × '*q*) *fset* **and** *P* :: '*q fset*
begin

sublocale *productive-horn TA* Δ Δ_ε *P* .

lemma *infer1*:
infer1 p (*fset bs*) = *set* (*productive-infer1-cont* Δ Δ_ε *p bs*)
unfolding *productive-infer1 productive-infer1-cont-def set-append Un-assoc*[*symmetric*]
unfolding *union-fset sorted-list-of-fset-simps Let-def set-append*
apply (*intro arg-cong2*[*of* - - - - (∪)])
subgoal
by (*simp add*: *image-def Bex-def map-val-of-list-simp*)
subgoal
apply (*auto simp*: *map-val-of-list-simp image-iff*)
apply (*metis ta-rule.sel*(*2*, *3*))
apply (*metis ta-rule.collapse*)
done
done

sublocale *l*: *horn-fset productive-rules P* (*TA* Δ Δ_ε) *sorted-list-of-fset P productive-infer1-cont* Δ Δ_ε
apply (*unfold-locales*)
using *infer1 productive-infer0 fset-of-list.rep-eq*
by *fastforce*+

lemmas *infer* = *l.infer0 l.infer1*
lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition *productive-cont-impl P* Δ Δ_ε =
horn-fset-impl.saturate-impl (*sorted-list-of-fset P*) (*productive-infer1-cont* Δ Δ_ε)

lemma *productive-cont-impl-sound*:
productive-cont-impl P Δ Δ_ε = *Some xs* $\Longrightarrow$ *fset xs* = *ta-productive-ind P* (*TA* Δ Δ_ε)
using *productive-rules-fset.saturate-impl-sound* **unfolding** *productive-cont-impl-def*
unfolding *productive-horn.productive-sound* **.**

lemma *productive-cont-impl-complete*:
productive-cont-impl P Δ Δ_ε $\neq$ *None*
proof −
have *finite* (*ta-productive-ind P* (*TA* Δ Δ_ε))
unfolding *ta-productive-ind* **by** *simp*
then show *?thesis* **unfolding** *productive-cont-impl-def*
by (*intro productive-rules-fset.saturate-impl-complete*)
(*auto simp*: *productive-horn.productive-sound*)
qed

lemma *productive-impl* [*code*]:
ta-productive P (*TA* Δ Δ_ε) = *the* (*productive-cont-impl P* Δ Δ_ε)
using *productive-cont-impl-complete*[*of P* Δ] *productive-cont-impl-sound*[*of P* Δ]
by (*auto simp add*: *ta-productive-ind fset-of-list-elem*)

9.6 Setup for the implementation of power set construction states

abbreviation *r-statesl r* ≡ *length* (*r-lhs-states r*)

definition *ps-reachable-states-list* **where**
ps-reachable-states-list mapp-r mapp-e f ps =
(*let R* = *filter* (λ *r*. *list-all2* (|∈|) (*r-lhs-states r*) *ps*)
(*case-option Nil id* (*Mapping.lookup mapp-r* (*f*, *length ps*))) *in*
let S = *map r-rhs R in*
S @ *concat* (*map* (*case-option Nil id* ∘ *Mapping.lookup mapp-e*) *S*))

lemma *ps-reachable-states-list-sound*:
assumes *length ps* = *n*
and *mapp-r*: *case-option Nil id* (*Mapping.lookup mapp-r* (*f*, *n*)) =
filter (λ*r*. *r-root r* = *f* ∧ *r-statesl r* = *n*) (*sorted-list-of-fset* Δ)
and *mapp-e*: $\bigwedge p$. *case-option Nil id* (*Mapping.lookup mapp-e p*) =
map snd (*filter* (λ *q*. *fst q* = *p*) (*sorted-list-of-fset* ($\Delta_\varepsilon|^+|$)))

shows *fset-of-list* (*ps-reachable-states-list mapp-r mapp-e f* (*map ex ps*)) =
ps-reachable-states (*TA* Δ Δ_ε) *f* (*map ex ps*) (**is** *?Ls* = *?Rs*)
proof −
have ∗: *length ps* = *n length* (*map ex ps*) = *n* **using** *assms* **by** *auto*
{fix *q* **assume** *q* |∈| *?Ls*
then obtain *qs p* **where** *TA-rule f qs p* |∈| Δ *length ps* = *length qs*
list-all2 (|∈|) *qs* (*map ex ps*) *p* = *q* ∨ (*p*, *q*) |∈| $\Delta_\varepsilon|^+|$
unfolding *ps-reachable-states-list-def Let-def comp-def assms*(*1*, *2*, *3*) ∗
by (*force simp add*: *fset-of-list-elem image-iff fBex-def*)
then have *q* |∈| *?Rs*
by (*force simp add*: *ps-reachable-states-fmember image-iff*)**}**
moreover
{fix *q* **assume** *q* |∈| *?Rs*
then obtain *qs p* **where** *TA-rule f qs p* |∈| Δ *length ps* = *length qs*
list-all2 (|∈|) *qs* (*map ex ps*) *p* = *q* ∨ (*p*, *q*) |∈| $\Delta_\varepsilon|^+|$
by (*auto simp add*: *ps-reachable-states-fmember list-all2-iff*)
then have *q* |∈| *?Ls*
unfolding *ps-reachable-states-list-def Let-def* ∗ *comp-def assms*(*2*, *3*)
by (*force simp add*: *fset-of-list-elem image-iff*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *rule-target-statesI*:
∃ *r* |∈| Δ. *r-rhs r* = *q* ⟹ *q* |∈| *rule-target-states* Δ
by *auto*

definition *ps-states-infer0-cont* :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* ⇒
('*q* × '*q*) *fset* ⇒ '*q FSet-Lex-Wrapper list* **where**
ps-states-infer0-cont Δ Δ_ε =
(*let sig* = *filter* (λ *r*. *r-lhs-states r* = []) (*sorted-list-of-fset* Δ) *in*
filter (λ *p*. *ex p* ≠ {||}) (*map* (λ *r*. *Wrapp* (*ps-reachable-states* (*TA* Δ Δ_ε)
(*r-root r*) [])) *sig*))

definition *ps-states-infer1-cont* :: ('*q* :: *linorder* , '*f* :: *linorder*) *ta-rule fset* ⇒ ('*q*
× '*q*) *fset* ⇒
'*q FSet-Lex-Wrapper* ⇒ '*q FSet-Lex-Wrapper fset* ⇒ '*q FSet-Lex-Wrapper list*
where
ps-states-infer1-cont Δ Δ_ε =
(*let sig* = *remdups* (*map* (λ *r*. (*r-root r*, *r-statesl r*)) (*filter* (λ *r*. *r-lhs-states r*
≠ []) (*sorted-list-of-fset* Δ))) *in*
let arities = *remdups* (*map snd sig*) *in*
let etr = *sorted-list-of-fset* ($\Delta_\varepsilon|^+|$) *in*
let mapp-r = *map-of-list* (λ *r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* Δ)
in
let mapp-e = *map-of-list fst snd etr in*
(λ *p bs*.
(*let states* = *sorted-list-of-fset* (*finsert p bs*) *in*
let arity-to-states-map = *Mapping.tabulate arities* (λ *n*. *list-of-permutation-element-n*

```
p n states) in
      let res = map (λ (f, n).
        map (λ s. let rules = the (Mapping.lookup mapp-r (f, n)) in
          Wrapp (fset-of-list (ps-reachable-states-list mapp-r mapp-e f (map ex s))))
          (the (Mapping.lookup arity-to-states-map n)))
         sig in
     filter (λ p. ex p ≠ {||}) (concat res))))

locale ps-states-fset =
  fixes Δ :: ('q :: linorder, 'f :: linorder) ta-rule fset and Δε :: ('q × 'q) fset
begin

sublocale ps-states-horn TA Δ Δε .

lemma infer0: infer0 = set (ps-states-infer0-cont Δ Δε)
  unfolding ps-states-horn.ps-construction-infer0
  unfolding ps-states-infer0-cont-def Let-def
  using ps-reachable-states-fmember
  by (auto simp add: image-def Ball-def Bex-def)
    (metis list-all2-Nil2 ps-reachable-states-fmember ta.sel(1) ta-rule.sel(1, 2))

lemma r-lhs-states-nConst:
  r-lhs-states r ≠ [] ⟹ r-statesl r ≠ 0 for r by auto


lemma filter-empty-conv':
  [] = filter P xs ⟷ (∀ x∈set xs. ¬ P x)
  by (metis filter-empty-conv)

lemma infer1:
  infer1 p (fset bs) = set (ps-states-infer1-cont Δ Δε p bs) (is ?Ls = ?Rs)
proof −
  let ?mapp-r = map-of-list (λr. (r-root r, r-statesl r)) (λx. x) (sorted-list-of-fset
Δ)
  let ?mapp-e = map-of-list fst snd (sorted-list-of-fset (Δε|+|))
  have mapr: case-option Nil id (Mapping.lookup ?mapp-r (f, n)) =
    filter (λr. r-root r = f ∧ r-statesl r = n) (sorted-list-of-fset Δ) for f n
   by (auto simp: map-val-of-list-simp image-iff filter-empty-conv' intro: filter-cong)
  have epsr: ⋀p. case-option Nil id (Mapping.lookup ?mapp-e p) =
    map snd (filter (λ q. fst q = p) (sorted-list-of-fset (Δε|+|)))
   by (auto simp: map-val-of-list-simp image-iff filter-empty-conv) metis
  have ∗: p ∈ set qs ⟹ x |∈| ps-reachable-states (TA Δ Δε) f (map ex qs) ⟹
   (∃ ps q. TA-rule f ps q |∈| Δ ∧ length ps = length qs) for x f qs
   by (auto simp: ps-reachable-states-fmember list-all2-conv-all-nth)
  {fix q assume q ∈ ?Ls
    then obtain f qss where sp: q = Wrapp (ps-reachable-states (TA Δ Δε) f
(map ex qss))
      ps-reachable-states (TA Δ Δε) f (map ex qss) ≠ {||} p ∈ set qss set qss ⊆
insert p (fset bs)
```

by (*auto simp add*: *ps-construction-infer1 ps-reachable-states-fmember*)
from *sp*(*2*, *3*) **obtain** *ps p'* **where** *r*: *TA-rule f ps p'* |∈| Δ *length ps* = *length qss* **using** ∗
by *blast*
then have *mem*: *qss* ∈ *set* (*list-of-permutation-element-n p* (*length ps*) (*sorted-list-of-fset* (*finsert p bs*))) **using** *sp*(*2*−)
by (*auto simp*: *list-of-permutation-element-n-iff*)
(*meson in-set-idx insertE set-list-subset-eq-nth-conv*)
then have *q* ∈ *?Rs* **using** *sp r*
unfolding *ps-construction-infer1 ps-states-infer1-cont-def Let-def*
apply (*simp add*: *lookup-tabulate ps-reachable-states-fmember image-iff*)
apply (*rule-tac x* = *f ps* → *p'* **in** *exI*)
apply (*auto simp*: *Bex-def ps-reachable-states-list-sound*[*OF - mapr epsr*] *intro*: *exI*[*of - qss*])
done}
moreover
{**fix** *q* **assume** *ass*: *q* ∈ *?Rs*
then obtain *r qss* **where** *r* |∈| Δ *r-lhs-states r* ≠ [] *qss* ∈ *set* (*list-of-permutation-element-n p* (*r-statesl r*) (*sorted-list-of-fset* (*finsert p bs*)))
q = *Wrapp* (*ps-reachable-states* (*TA* Δ Δ_ε) (*r-root r*) (*map ex qss*))
unfolding *ps-states-infer1-cont-def Let-def*
by (*auto simp add*: *lookup-tabulate ps-reachable-states-fmember image-iff ps-reachable-states-list-sound*[*OF - mapr epsr*] *split*: *if-splits*)
moreover have *q* ≠ *Wrapp* {||} **using** *ass*
by (*auto simp*: *ps-states-infer1-cont-def Let-def*)
ultimately have *q* ∈ *?Ls* **unfolding** *ps-construction-infer1*
apply (*auto simp*: *list-of-permutation-element-n-iff intro*!: *exI*[*of - r-root r*] *exI*[*of - qss*])
apply (*metis in-set-idx*)
done}
ultimately show *?thesis* **by** *blast*
qed

sublocale *l*: *horn-fset ps-states-rules* (*TA* Δ Δ_ε) *ps-states-infer0-cont* Δ Δ_ε *ps-states-infer1-cont* Δ Δ_ε
apply (*unfold-locales*)
using *infer0 infer1*
by *fastforce*+

lemmas *infer* = *l.infer0 l.infer1*
lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition *ps-states-fset-impl* Δ Δ_ε =
horn-fset-impl.saturate-impl (*ps-states-infer0-cont* Δ Δ_ε) (*ps-states-infer1-cont*

$\Delta\ \Delta_\varepsilon$)

lemma *ps-states-fset-impl-sound*:
assumes *ps-states-fset-impl* $\Delta\ \Delta_\varepsilon$ = *Some xs*
shows *xs* = *ps-states* (*TA* $\Delta\ \Delta_\varepsilon$)
using *ps-states-fset.saturate-impl-sound*[*OF assms*[*unfolded ps-states-fset-impl-def*]]
using *ps-states-horn.ps-states-sound*[*of TA* $\Delta\ \Delta_\varepsilon$]
by (*auto simp*: *fset-of-list-elem ps-states.rep-eq fset-of-list.rep-eq*)

lemma *ps-states-fset-impl-complete*:
ps-states-fset-impl $\Delta\ \Delta_\varepsilon \neq$ *None*
proof −
let *?R* = *ps-states* (*TA* $\Delta\ \Delta_\varepsilon$)
let *?S* = *horn.saturate* (*ps-states-rules* (*TA* $\Delta\ \Delta_\varepsilon$))
have *?S* $\subseteq$ *fset ?R*
using *ps-states-horn.ps-states-sound*
by (*simp add*: *ps-states-horn.ps-states-sound ps-states.rep-eq*)
from *finite-subset*[*OF this*] **show** *?thesis*
unfolding *ps-states-fset-impl-def*
by (*intro ps-states-fset.saturate-impl-complete*) *simp*
qed

lemma *ps-ta-impl* [*code*]:
ps-ta (*TA* $\Delta\ \Delta_\varepsilon$) =
(*let xs* = *the* (*ps-states-fset-impl* $\Delta\ \Delta_\varepsilon$) *in*
TA (*ps-rules* (*TA* $\Delta\ \Delta_\varepsilon$) *xs*) {||})
using *ps-states-fset-impl-complete*
using *ps-states-fset-impl-sound*
unfolding *ps-ta-def Let-def*
by (*metis option.exhaust-sel*)

lemma *ps-reg-impl* [*code*]:
ps-reg (*Reg Q* (*TA* $\Delta\ \Delta_\varepsilon$)) =
(*let xs* = *the* (*ps-states-fset-impl* $\Delta\ \Delta_\varepsilon$) *in*
Reg (*ffilter* (λ *S*. *Q* |$\cap$| *ex S* $\neq$ {||}) *xs*)
(*TA* (*ps-rules* (*TA* $\Delta\ \Delta_\varepsilon$) *xs*) {||}))
using *ps-states-fset-impl-complete*[*of* $\Delta\ \Delta_\varepsilon$]
using *ps-states-fset-impl-sound*[*of* $\Delta\ \Delta_\varepsilon$]
unfolding *ps-reg-def ps-ta-*Q_f*-def Let-def*
unfolding *ps-ta-def Let-def*
using *eq-ffilter* **by** *auto*

lemma *prod-ta-zip* [*code*]:
prod-ta-rules ($\mathcal{A}$ :: (*'q1* :: *linorder*, *'f* :: *linorder*) *ta*) ($\mathcal{B}$ :: (*'q2* :: *linorder*, *'f* :: *linorder*) *ta*) =
(*let sig* = *sorted-list-of-fset* (*ta-sig* $\mathcal{A}$ |$\cap$| *ta-sig* $\mathcal{B}$) *in*
let mapA = *map-of-list* (λ*r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* (*rules* $\mathcal{A}$)) *in*
let mapB = *map-of-list* (λ*r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* (*rules*

```
\<B>)) in
    let merge = (λ (ra, rb). TA-rule (r-root ra) (zip (r-lhs-states ra) (r-lhs-states
rb)) (r-rhs ra, r-rhs rb)) in
     fset-of-list (
     concat (map (λ (f, n). map merge
         (List.product (the (Mapping.lookup mapA (f, n))) (the (Mapping.lookup
mapB (f, n))))) sig)))
 (is ?Ls = ?Rs)
proof −
  have [simp]: distinct (sorted-list-of-fset (ta-sig \<A>))  distinct (sorted-list-of-fset
(ta-sig \<B>))
    by (simp-all add: distinct-sorted-list-of-fset)
  have ∗: sort (remdups (map (λr. (r-root r, r-statesl r)) (sorted-list-of-fset (rules
\<A>)))) = sorted-list-of-fset (ta-sig \<A>)
   sort (remdups (map (λr. (r-root r, r-statesl r)) (sorted-list-of-fset (rules \<B>))))
= sorted-list-of-fset (ta-sig \<B>)
    by (auto simp: ta-sig-def sorted-list-of-fset-fimage-dist)
  {fix r assume ass: r |∈| ?Ls
   then obtain f qs q where [simp]: r = f qs → q by auto
   then have (f, length qs) |∈| ta-sig \<A> |∩| ta-sig \<B> using ass by auto
   then have r |∈| ?Rs using ass unfolding map-val-of-list-tabulate-conv ∗
    by (auto simp: Let-def fset-of-list-elem image-iff case-prod-beta lookup-tabulate
intro!: bexI[of - (f, length qs)])
       (metis (no-types, lifting) length-map ta-rule.sel(1 − 3) zip-map-fst-snd)}
   moreover
  {fix r assume ass: r |∈| ?Rs then have r |∈| ?Ls unfolding map-val-of-list-tabulate-conv
∗
      by (auto simp: fset-of-list-elem finite-Collect-prod-ta-rules lookup-tabulate)
        (metis ta-rule.collapse)}
  ultimately show ?thesis by blast
qed



end
theory RR2-Infinite-Q-infinity
  imports RR2-Infinite
begin



lemma if-cong':
  b = c ⟹ x = u ⟹ y = v ⟹ (if b then x else y) = (if c then u else v)
  by auto


fun ta-der-strict :: ('q,'f) ta ⇒ ('f,'q) term ⇒ 'q fset where
  ta-der-strict \<A> (Var q) = {|q|}
| ta-der-strict \<A> (Fun f ts) = {| q' | q' q qs. TA-rule f qs q |∈| rules \<A> ∧ (q = q'
```

$\vee$ $(q,\ q')$ $|\in|$ $(eps\ \mathcal{A})|^+|)$ $\wedge$
$length\ qs = length\ ts \wedge (\forall\ i < length\ ts.\ qs\ !\ i\ |\in|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ (ts\ !\ i))|\}$

lemma *ta-der-strict-Var*:
$q\ |\in|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ (Var\ x) \longleftrightarrow x = q$
unfolding *ta-der.simps* **by** *auto*

lemma *ta-der-strict-Fun*:
$q\ |\in|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ (Fun\ f\ ts) \longleftrightarrow (\exists\ ps\ p.\ TA\text{-}rule\ f\ ps\ p\ |\in|\ (rules\ \mathcal{A}) \wedge$
$(p = q \vee (p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+|) \wedge length\ ps = length\ ts \wedge$
$(\forall\ i < length\ ts.\ ps\ !\ i\ |\in|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ (ts\ !\ i)))$ (**is** *?Ls* $\longleftrightarrow$ *?Rs*)
unfolding *ta-der-strict.simps*
by (*intro iffI fCollect-memberI finite-Collect-less-eq*[*OF - finite-eps*[*of* $\mathcal{A}$]]) *auto*

declare *ta-der-strict.simps*[*simp del*]
lemmas *ta-der-strict-simps* [*simp*] = *ta-der-strict-Var ta-der-strict-Fun*

lemma *ta-der-strict-sub-ta-der*:
$ta\text{-}der\text{-}strict\ \mathcal{A}\ t\ |\subseteq|\ ta\text{-}der\ \mathcal{A}\ t$
proof (*induct t*)
case (*Fun f ts*)
then show *?case*
by *auto* (*metis fsubsetD nth-mem*)+
qed *auto*

lemma *ta-der-strict-ta-der-eq-on-ground*:
assumes*ground t*
shows $ta\text{-}der\ \mathcal{A}\ t = ta\text{-}der\text{-}strict\ \mathcal{A}\ t$
proof
{fix q **assume** $q\ |\in|\ ta\text{-}der\ \mathcal{A}\ t$ **then have** $q\ |\in|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ t$ **using** *assms*
proof (*induct t arbitrary*: *q*)
case (*Fun f ts*)
then show *?case* **apply** *auto*
using *nth-mem* **by** *blast*+
qed *auto***}**
then show $ta\text{-}der\ \mathcal{A}\ t\ |\subseteq|\ ta\text{-}der\text{-}strict\ \mathcal{A}\ t$
by *auto*
next
show $ta\text{-}der\text{-}strict\ \mathcal{A}\ t\ |\subseteq|\ ta\text{-}der\ \mathcal{A}\ t$ **using** *ta-der-strict-sub-ta-der* **.**
qed

lemma *ta-der-to-ta-strict*:
assumes $q\ |\in|\ ta\text{-}der\ A\ C\langle Var\ p\rangle$ **and** *ground-ctxt C*
shows $\exists\ q'.\ (p = q' \vee (p,\ q')\ |\in|\ (eps\ A)|^+|) \wedge q\ |\in|\ ta\text{-}der\text{-}strict\ A\ C\langle Var\ q'\rangle$
using *assms*
proof (*induct C arbitrary*: *q p*)
case (*More f ss C ts*)
from *More*(*2*) **obtain** *qs* q' **where**
r: $TA\text{-}rule\ f\ qs\ q'\ |\in|\ rules\ A\ length\ qs = Suc\ (length\ ss + length\ ts)\ q' = q \vee$

$(q', q) \mid\in\mid (eps\ A)^{+}\mid$ **and**
rec: $\forall\ i < length\ qs.\ qs\ !\ i \mid\in\mid ta\text{-}der\ A\ ((ss\ @\ C\langle Var\ p\rangle\ \#\ ts)\ !\ i)$
by *auto*
from *More*(*1*)[*of qs* ! *length ss p*] *More*(*3*) *rec r*(*2*) **obtain** q'' **where**
mid: $(p = q'' \vee (p, q'') \mid\in\mid (eps\ A)^{+}\mid) \wedge qs\ !\ length\ ss \mid\in\mid ta\text{-}der\text{-}strict\ A\ C\langle Var\ q''\rangle$
by *auto* (*metis length-map less-add-Suc1 nth-append-length*)
then have $\forall\ i < length\ qs.\ qs\ !\ i \mid\in\mid ta\text{-}der\text{-}strict\ A\ ((ss\ @\ C\langle Var\ q''\rangle\ \#\ ts)\ !\ i)$
using *rec r*(*2*) *More*(*3*)
using *ta-der-strict-ta-der-eq-on-ground*[*of* - *A*]
by (*auto simp*: *nth-append-Cons all-Suc-conv split*:*if-splits cong*: *if-cong*′)
then show *?case* **using** *rec r conjunct1*[*OF mid*]
by (*rule-tac* $x = q''$ **in** *exI*, *auto intro*!: *exI*[*of* - q'] *exI*[*of* - *qs*])
qed *auto*

fun *root-ctxt* **where**
root-ctxt (*More f ss C ts*) = *f*
| *root-ctxt* $\square$ = *undefined*

lemma *root-to-root-ctxt* [*simp*]:
assumes $C \neq \square$
shows *fst* (*the* (*root* $C\langle t\rangle$)) $\longleftrightarrow$ *root-ctxt C*
using *assms* **by** (*cases C*) *auto*

inductive-set *Q-inf* **for** $\mathcal{A}$ **where**
trans: $(p, q) \in Q\text{-}inf\ \mathcal{A} \Longrightarrow (q, r) \in Q\text{-}inf\ \mathcal{A} \Longrightarrow (p, r) \in Q\text{-}inf\ \mathcal{A}$
| *rule*: $(None, Some\ f)\ qs \to q \mid\in\mid rules\ \mathcal{A} \Longrightarrow i < length\ qs \Longrightarrow (qs\ !\ i, q) \in Q\text{-}inf\ \mathcal{A}$
| *eps*: $(p, q) \in Q\text{-}inf\ \mathcal{A} \Longrightarrow (q, r) \mid\in\mid eps\ \mathcal{A} \Longrightarrow (p, r) \in Q\text{-}inf\ \mathcal{A}$

abbreviation $Q\text{-}inf\text{-}e\ \mathcal{A} \equiv \{q \mid p\ q.\ (p, p) \in Q\text{-}inf\ \mathcal{A} \wedge (p, q) \in Q\text{-}inf\ \mathcal{A}\}$

lemma *Q-inf-states-ta-states*:
assumes $(p, q) \in Q\text{-}inf\ \mathcal{A}$
shows $p \mid\in\mid \mathcal{Q}\ \mathcal{A}\ q \mid\in\mid \mathcal{Q}\ \mathcal{A}$
using *assms* **by** (*induct*) (*auto simp*: *rule-statesD eps-statesD*)

lemma *Q-inf-finite*:
finite (*Q-inf* $\mathcal{A}$) *finite* (*Q-inf-e* $\mathcal{A}$)
proof −
have $*$: *Q-inf* $\mathcal{A} \subseteq fset\ (\mathcal{Q}\ \mathcal{A} \mid\times\mid \mathcal{Q}\ \mathcal{A})$ *Q-inf-e* $\mathcal{A} \subseteq fset\ (\mathcal{Q}\ \mathcal{A})$
by (*auto simp add*: *Q-inf-states-ta-states*(*1*, *2*) *subrelI*)
show *finite* (*Q-inf* $\mathcal{A}$)
by (*intro finite-subset*[*OF* $*$(*1*)]) *simp*
show *finite* (*Q-inf-e* $\mathcal{A}$)
by (*intro finite-subset*[*OF* $*$(*2*)]) *simp*

qed

context
includes *fset.lifting*
begin
lift-definition *fQ-inf* :: ('*a*, '*b option* × '*c option*) *ta* ⇒ ('*a* × '*a*) *fset* **is** *Q-inf*
by (*simp add*: *Q-inf-finite*(*1*))
lift-definition *fQ-inf-e* :: ('*a*, '*b option* × '*c option*) *ta* ⇒ '*a fset* **is** *Q-inf-e*
using *Q-inf-finite*(*2*) **.**
end

lemma *Q-inf-ta-eps-Q-inf*:
assumes (*p*, *q*) ∈ *Q-inf* $\mathcal{A}$ **and** (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)|$^+$|
shows (*p*, *q'*) ∈ *Q-inf* $\mathcal{A}$ **using** *assms*(*2*, *1*)
by (*induct rule*: *ftrancl-induct*) (*auto simp add*: *Q-inf.eps*)

lemma *lhs-state-rule*:
assumes (*p*, *q*) ∈ *Q-inf* $\mathcal{A}$
shows ∃ *f qs r*. (*None*, *Some f*) *qs* → *r* |∈| *rules* $\mathcal{A}$ ∧ *p* |∈| *fset-of-list qs*
using *assms* **by** *induct* (*force intro*: *nth-mem*)+

lemma *Q-inf-reach-state-rule*:
assumes (*p*, *q*) ∈ *Q-inf* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows ∃ *ss ts f C*. *q* |∈| *ta-der* $\mathcal{A}$ (*More* (*None*, *Some f*) *ss C ts*)⟨*Var p*⟩ ∧ *ground-ctxt* (*More* (*None*, *Some f*) *ss C ts*)
(**is** ∃ *ss ts f C*. *?P ss ts f C q p*)
using *assms*
proof (*induct*)
case (*trans p q r*)
then obtain *f1 f2 ss1 ts1 ss2 ts2 C1 C2* **where**
C: *?P ss1 ts1 f1 C1 q p ?P ss2 ts2 f2 C2 r q* **by** *blast*
then show *?case*
apply (*rule-tac x* = *ss2* **in** *exI*, *rule-tac x* = *ts2* **in** *exI*, *rule-tac x* = *f2* **in** *exI*,
rule-tac x = *C2* ∘$_c$ (*More* (*None*, *Some f1*) *ss1 C1 ts1*) **in** *exI*)
apply (*auto simp del*: *intp-actxt.simps*)
apply (*metis Subterm-and-Context.ctxt-ctxt-compose actxt-compose.simps*(*2*) *ta-der-ctxt*)
done
next
case (*rule f qs q i*)
have ∀ *i* < *length qs*. ∃ *t*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *t* ∧ *ground t*
using *rule*(*1*, *2*) *fset-mp*[*OF rule*(*3*), *of qs* ! *i* **for** *i*]
by *auto* (*meson fnth-mem rule-statesD*(*4*) *ta-reachableE*)
then obtain *ts* **where** *wit*: *length ts* = *length qs*
∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*) ∧ *ground* (*ts* ! *i*)
using *Ex-list-of-length-P*[*of length qs* λ *x i*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ *x* ∧ *ground x*]
by *blast*
{fix *j* **assume** *j* < *length qs*

```
    then have qs ! j |∈| ta-der 𝒜 ((take i ts @ Var (qs ! i) # drop (Suc i) ts) ! j)
      using wit by (cases j < i) (auto simp: min-def nth-append-Cons)}
  then have ∀ i < length qs. qs ! i |∈| (map (ta-der 𝒜) (take i ts @ Var (qs ! i)
# drop (Suc i) ts)) ! i
    using wit rule(2) by (auto simp: nth-append-Cons)
  then have res: q |∈| ta-der 𝒜 (Fun (None, Some f) (take i ts @ Var (qs ! i) #
drop (Suc i) ts))
    using rule(1, 2) wit by (auto simp: min-def nth-append-Cons intro!: exI[of -
q] exI[of - qs])
  then show ?case using rule(1, 2) wit
    apply (rule-tac x = take i ts in exI, rule-tac x = drop (Suc i) ts in exI)
    apply (auto simp: take-map drop-map  dest!: in-set-takeD in-set-dropD simp
del: ta-der-simps intro!: exI[of - f] exI[of - Hole])
    apply (metis all-nth-imp-all-set)+
    done
next
  case (eps p q r)
  then show ?case by (meson r-into-rtrancl ta-der-eps)
qed

lemma rule-target-Q-inf:
  assumes (None, Some f) qs → q' |∈| rules 𝒜 and i < length qs
   shows (qs ! i, q') ∈ Q-inf 𝒜 using assms
  by (intro rule) auto

lemma rule-target-eps-Q-inf:
  assumes (None, Some f) qs → q' |∈| rules 𝒜 (q', q) |∈| (eps 𝒜)|^+|
     and i < length qs
   shows (qs ! i, q) ∈ Q-inf 𝒜
  using assms(2, 1, 3) by (induct rule: ftrancl-induct) (auto intro: rule eps)


lemma step-in-Q-inf:
  assumes q |∈| ta-der-strict 𝒜 (map-funs-term (λf. (None, Some f)) (Fun f (ss
@ Var p # ts)))
    shows (p, q) ∈ Q-inf 𝒜
  using assms rule-target-eps-Q-inf[of f - - 𝒜 q] rule-target-Q-inf[of f - q 𝒜]
  by (auto simp: comp-def nth-append-Cons split!: if-splits)


lemma ta-der-Q-inf:
  assumes q |∈| ta-der-strict 𝒜 (map-funs-term (λf. (None, Some f)) (C⟨Var p⟩))
and C ≠ Hole
  shows (p, q) ∈ Q-inf 𝒜 using assms
proof (induct C arbitrary: q)
  case (More f ss C ts)
  then show ?case
  proof (cases C = Hole)
    case True
```

```
    then show ?thesis using More(2) by (auto simp: step-in-Q-inf)
  next
    case False
     then obtain q' where q: q' |∈| ta-der-strict 𝒜 (map-funs-term (λf. (None, Some f)) C⟨Var p⟩)
      q |∈| ta-der-strict 𝒜 (map-funs-term (λf. (None, Some f)) (Fun f (ss @ Var q' # ts)))
       using More(2) length-map

       by (auto simp: comp-def nth-append-Cons split: if-splits cong: if-cong')
         (smt (verit) nat-neq-iff nth-map ta-der-strict-simps)+
    have (p, q') ∈ Q-inf 𝒜 using More(1)[OF q(1) False] .
    then show ?thesis using step-in-Q-inf[OF q(2)] by (auto intro: trans)
  qed
qed auto

lemma Q-inf-e-infinite-terms-res:
  assumes q ∈ Q-inf-e 𝒜 and 𝒬 𝒜 |⊆| ta-reachable 𝒜
  shows infinite {t. q |∈| ta-der 𝒜 (term-of-gterm t) ∧ fst (groot-sym t) = None}
proof −
  let ?P =λ u. ground u ∧ q |∈| ta-der 𝒜 u ∧ fst (fst (the (root u))) = None
  have groot[simp]: fst (fst (the (root (term-of-gterm t)))) = fst (groot-sym t) for t by (cases t) auto
  have [simp]: C ≠ □ ⟹ fst (fst (the (root C⟨t⟩))) = fst (root-ctxt C) for C t by (cases C) auto
  from assms(1) obtain p where cycle: (p, p) ∈ Q-inf 𝒜 (p, q) ∈ Q-inf 𝒜 by auto
  from Q-inf-reach-state-rule[OF cycle(1) assms(2)] obtain C where
    ctxt: C ≠ □ ground-ctxt C and reach: p |∈| ta-der 𝒜 C⟨Var p⟩
    by blast
  obtain C2 where
    closing-ctxt: C2 ≠ □ ground-ctxt C2 fst (root-ctxt C2) = None and cl-reach: q |∈| ta-der 𝒜 C2⟨Var p⟩
   by (metis (full-types) Q-inf-reach-state-rule[OF cycle(2) assms(2)] actxt.distinct(1) fst-conv root-ctxt.simps(1))
  from assms(2) obtain t where t: p |∈| ta-der 𝒜 t and gr-t: ground t
    by (meson Q-inf-states-ta-states(1) cycle(1) fsubsetD ta-reachableE)
  let ?terms = λ n. (C ^ Suc n)⟨t⟩ let ?S = {?terms n | n. p |∈| ta-der 𝒜 (?terms n) ∧ ground (?terms n)}
  have ground (?terms n) for n using ctxt(2) gr-t by auto
  moreover have p |∈| ta-der 𝒜 (?terms n) for n using reach t(1)
    by (auto simp: ta-der-ctxt) (meson ta-der-ctxt ta-der-ctxt-n-loop)
  ultimately have inf: infinite ?S using ctxt-comp-n-lower-bound[OF ctxt(1)]
    using no-upper-bound-infinite[of - depth, of ?S] by blast
  from infinite-inj-image-infinite[OF this] have inf:infinite (ctxt-apply-term C2 ‘ ?S)
    by (smt (verit) ctxt-eq inj-on-def)
  {fix u assume u ∈ (ctxt-apply-term C2 ‘ ?S)
    then have ?P u unfolding image-Collect using closing-ctxt cl-reach
```

by (*auto simp*: *ta-der-ctxt*)**}**
from *this* **have** *inf*: *infinite* {*u*. *ground u* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *u* ∧ *fst* (*fst* (*the* (*root u*))) = *None*}
by (*intro infinite-super*[*OF* - *inf*] *subsetI*) *fast*
have *inf*: *infinite* (*gterm-of-term* ‘ {*u*. *ground u* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *u* ∧ *fst* (*fst* (*the* (*root u*))) = *None*})
by (*intro infinite-inj-image-infinite*[*OF inf*] *gterm-of-term-inj*) *auto*
show *?thesis*
by (*intro infinite-super*[*OF* - *inf*]) (*auto dest*: *groot-sym-gterm-of-term*)
qed

lemma *gfun-at-after-hole-pos*:
assumes *ghole-pos C* ≤$_p$ *p*
shows *gfun-at C*⟨*t*⟩$_G$ *p* = *gfun-at t* (*p* −$_p$ *ghole-pos C*) **using** *assms*
proof (*induct C arbitrary*: *p*)
case (*GMore f ss C ts*) **then show** *?case*
by (*cases p*) *auto*
qed *auto*

lemma *pos-diff-0* [*simp*]: *p* −$_p$ *p* = []
by (*auto simp*: *pos-diff-def*)

lemma *Max-suffI*: *finite A* ⟹ *A* = *B* ⟹ *Max A* = *Max B*
by (*intro Max-eq-if*) *auto*

lemma *nth-args-depth-eqI*:
assumes *length ss* = *length ts*
and ⋀ *i*. *i* < *length ts* ⟹ *depth* (*ss* ! *i*) = *depth* (*ts* ! *i*)
shows *depth* (*Fun f ss*) = *depth* (*Fun g ts*)
proof −
from *assms*(*1*, *2*) **have** *depth* ‘ *set ss* = *depth* ‘ *set ts*
using *nth-map-conv*[*OF assms*(*1*), *of depth depth*]
by (*simp flip*: *set-map*)
from *Max-suffI*[*OF* - *this*] **show** *?thesis* **using** *assms*(*1*)
by (*cases ss*; *cases ts*) *auto*
qed

lemma *subt-at-ctxt-apply-hole-pos* [*simp*]: $C\langle s\rangle$ |- *hole-pos* $C = s$
by (*induct* C) *auto*

lemma *ctxt-at-pos-ctxt-apply-hole-poss* [*simp*]: *ctxt-at-pos* $C\langle s\rangle$ (*hole-pos* C) $= C$
by (*induct* C) *auto*

abbreviation *map-funs-ctxt* $f \equiv$ *map-ctxt* f $(\lambda\ x.\ x)$
lemma *map-funs-term-ctxt-apply* [*simp*]:
map-funs-term f $C\langle s\rangle$ = (*map-funs-ctxt* f C)$\langle$*map-funs-term* f $s\rangle$
by (*induct* C) *auto*

lemma *map-funs-term-ctxt-decomp*:
assumes *map-funs-term* fg $t = C\langle s\rangle$
shows $\exists$ D u. C = *map-funs-ctxt* fg $D \wedge s$ = *map-funs-term* fg $u \wedge t = D\langle u\rangle$
using *assms*
proof (*induct* C *arbitrary*: t)
case *Hole*
show *?case*
by (*rule exI*[*of* - *Hole*], *rule exI*[*of* - t], *insert Hole*, *auto*)
next
case (*More* g *bef* C *aft*)
from *More*(*2*) **obtain** f *ts* **where** t: $t = Fun\ f\ ts$ **by** (*cases* t, *auto*)
from *More*(*2*)[*unfolded* t] **have** f: $fg\ f = g$ **and** *ts*: *map* (*map-funs-term* fg) *ts*
= *bef* @ $C\langle s\rangle$ # *aft* (**is** *?ts* = *?bca*) **by** *auto*
from *ts* **have** *length ?ts* = *length ?bca* **by** *auto*
then have *len*: *length ts* = *length ?bca* **by** *auto*
note *id* = *ts*[*unfolded map-nth-eq-conv*[*OF len*], *THEN spec*, *THEN mp*]
let *?i* = *length bef*
from *len* **have** *i*: *?i* < *length ts* **by** *auto*
from *id*[*of ?i*] **have** *map-funs-term fg* (*ts* ! *?i*) = $C\langle s\rangle$ **by** *auto*
from *More*(*1*)[*OF this*] **obtain** D u **where** D: C = *map-funs-ctxt* fg D **and**
u: s = *map-funs-term* fg u **and** *id*: *ts* ! *?i* = $D\langle u\rangle$ **by** *auto*
from *ts* **have** *take ?i ?ts* = *take ?i ?bca* **by** *simp*
also have ... = *bef* **by** *simp*
finally have *bef*: *map* (*map-funs-term fg*) (*take ?i ts*) = *bef* **by** (*simp add*:
take-map)
from *ts* **have** *drop* (*Suc ?i*) *?ts* = *drop* (*Suc ?i*) *?bca* **by** *simp*
also have ... = *aft* **by** *simp*
finally have *aft*: *map* (*map-funs-term fg*) (*drop* (*Suc ?i*) *ts*) = *aft* **by** (*simp*
add:*drop-map*)
let *?bda* = *take ?i ts* @ $D\langle u\rangle$ # *drop* (*Suc ?i*) *ts*
show *?case*
proof (*rule exI*[*of* - *More f* (*take ?i ts*) D (*drop* (*Suc ?i*) *ts*)],
rule exI[*of* - u], *simp add*: u f D *bef aft* t)
have *ts* = *take ?i ts* @ *ts* ! *?i* # *drop* (*Suc ?i*) *ts*
by (*rule id-take-nth-drop*[*OF i*])
also have ... = *?bda* **by** (*simp add*: *id*)
finally show *ts* = *?bda* **.**
qed

qed

lemma *prod-automata-from-none-root-dec*:
assumes *gta-lang* Q $\mathcal{A} \subseteq \{gpair\ s\ t |\ s\ t.\ funas\text{-}gterm\ s \subseteq \mathcal{F} \wedge funas\text{-}gterm\ t \subseteq \mathcal{F}\}$
and q $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm* t) **and** *fst* (*groot-sym* t) $=$ *None*
and $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* $\mathcal{A}$ **and** q $|\in|$ *ta-productive* Q $\mathcal{A}$
shows $\exists\ u.\ t =$ *gterm-to-None-Some* $u \wedge$ *funas-gterm* $u \subseteq \mathcal{F}$
proof $-$
have $*$: *gfun-at* t $[] =$ *Some* (*groot-sym* t) **by** (*cases* t) *auto*
from *assms*(*4*, *5*) **obtain** C q_f **where** *ctxt*: *ground-ctxt* C **and**
fin: q_f $|\in|$ *ta-der* $\mathcal{A}$ $C\langle Var\ q\rangle$ q_f $|\in|$ Q
by (*auto simp*: *ta-productive-def'*[*OF assms*(*4*)])
then obtain s v **where** *gp*: *gpair* s $v = (gctxt\text{-}of\text{-}ctxt\ C)\langle t\rangle_G$ **and**
funas: *funas-gterm* $v \subseteq \mathcal{F}$
using *assms*(*1*, *2*) *gta-langI*[*OF fin*(*2*), *of* $\mathcal{A}$ (*gctxt-of-ctxt* C)$\langle t\rangle_G$]
by (*auto simp*: *ta-der-ctxt ground-gctxt-of-ctxt-apply-gterm*)
from *gp* **have** *mem*: *hole-pos* $C \in$ *gposs* $s \cup$ *gposs* v
by *auto* (*metis Un-iff ctxt ghole-pos-in-apply gposs-of-gpair ground-hole-pos-to-ghole*)
from *this* **have** *hole-pos* $C \notin$ *gposs* s **using** *assms*(*3*)
using *arg-cong*[*OF gp*, *of* $\lambda\ t.$ *gfun-at* t (*hole-pos* C)]
using *ground-hole-pos-to-ghole*[*OF ctxt*]
using *gfun-at-after-hole-pos*[*OF position-less-refl*, *of gctxt-of-ctxt* C]
by (*auto simp*: *gfun-at-gpair* $*$ *split*: *if-splits*)
(*metis fstI gfun-at-None-ngposs-iff*)+
from *subst-at-gpair-nt-poss-None-Some*[*OF - this*, *of* v] *this*
have $t =$ *gterm-to-None-Some* (*gsubt-at* v (*hole-pos* C)) $\wedge$ *funas-gterm* (*gsubt-at* v (*hole-pos* C)) $\subseteq \mathcal{F}$
using *funas mem funas-gterm-gsubt-at-subseteq*
by (*auto simp*: *gp intro*!: *exI*[*of - gsubt-at* v (*hole-pos* C)])
(*metis ctxt ground-hole-pos-to-ghole gsubt-at-gctxt-apply-ghole*)
then show *?thesis* **by** *blast*
qed

lemma *infinite-set-dec-infinite*:
assumes *infinite* S **and** $\bigwedge\ s.\ s \in S \Longrightarrow \exists\ t.\ f\ t = s \wedge P\ t$
shows *infinite* $\{t \mid t\ s.\ s \in S \wedge f\ t = s \wedge P\ t\}$ (**is** *infinite ?T*)
proof (*rule ccontr*)
assume *ass*: $\neg$ *infinite ?T*
have $S \subseteq f$ ‘ $\{x\ .\ P\ x\}$ **using** *assms*(*2*) **by** *auto*
then show *False* **using** *ass assms*(*1*)
by (*auto simp*: *subset-image-iff*)
(*metis* (*mono-tags*, *lifting*) *Ball-Collect finite-imageI image-eqI infinite-super*)
qed

lemma *Q-inf-exec-impl-Q-inf*:
assumes *gta-lang Q* $\mathcal{A} \subseteq \{$*gpair s t*$|$ *s t. funas-gterm s* $\subseteq$ *fset* $\mathcal{F} \wedge$ *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}\}$
and $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-productive Q* $\mathcal{A}$
and *q* $\in$ *Q-inf-e* $\mathcal{A}$
shows *q* $|\in|$ *Q-infty* $\mathcal{A}$ $\mathcal{F}$
proof −
let *?S* = $\{$*t. q* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) $\wedge$ *fst* (*groot-sym t*) = *None*$\}$
let *?P* = λ *t. funas-gterm t* $\subseteq$ *fset* $\mathcal{F} \wedge$ *q* $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some t*))
let *?F* = (λ(*f*, *n*). ((*None*, *Some f*), *n*)) $|`|$ $\mathcal{F}$
from *Q-inf-e-infinite-terms-res*[*OF assms*(*4*, *2*)] **have** *inf*: *infinite ?S* **by** *auto*
{fix *t* **assume** *t* $\in$ *?S*
then have $\exists$ *u. t = gterm-to-None-Some u* $\wedge$ *funas-gterm u* $\subseteq$ *fset* $\mathcal{F}$
using *prod-automata-from-none-root-dec*[*OF assms*(*1*)] *assms*(*2*, *3*)
using *fin-mono* **by** *fastforce***}**
then show *?thesis* **using** *infinite-set-dec-infinite*[*OF inf*, *of gterm-to-None-Some ?P*]
by (*auto simp*: *Q-infty-fmember*) *blast*
qed

lemma *Q-inf-impl-Q-inf-exec*:
assumes *q* $|\in|$ *Q-infty* $\mathcal{A}$ $\mathcal{F}$
shows *q* $\in$ *Q-inf-e* $\mathcal{A}$
proof −
let *?t-of-g* = λ *t. term-of-gterm t* :: ('*b option* × '*b option*, '*a*) *term*
let *?t-og-g2* = λ *t. term-of-gterm t* :: ('*b*, '*a*) *term*
let *?inf* = (*?t-og-g2* :: '*b gterm* ⇒ ('*b*, '*a*) *term*) ‘ $\{$*t* $|$*t. funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *q* $|\in|$ *ta-der* $\mathcal{A}$ (*?t-of-g* (*gterm-to-None-Some t*))$\}$
obtain *n* **where** *card-st*: *fcard* ($\mathcal{Q}$ $\mathcal{A}$) < *n* **by** *blast*
from *assms*(*1*) **have** *infinite* $\{$*t* $|$*t. funas-gterm t* $\subseteq$ *fset* $\mathcal{F} \wedge$ *q* $|\in|$ *ta-der* $\mathcal{A}$ (*?t-of-g* (*gterm-to-None-Some t*))$\}$
unfolding *Q-infty-def* **by** *blast*
from *infinite-inj-image-infinite*[*OF this*, *of ?t-og-g2*] **have** *inf*: *infinite ?inf* **using** *inj-term-of-gterm* **by** *blast*
{fix *s* **assume** *s* $\in$ *?inf* **then have** *ground s funas-term s* $\subseteq$ *fset* $\mathcal{F}$ **by** (*auto simp*: *funas-term-of-gterm-conv subsetD*)**}**
from *infinte-no-depth-limit*[*OF inf*, *of fset* $\mathcal{F}$] *this* **obtain** *u* **where**
funas: *funas-gterm u* $\subseteq$ *fset* $\mathcal{F}$ **and** *card-d*: *n* < *depth* (*?t-og-g2 u*) **and** *reach*: *q* $|\in|$ *ta-der* $\mathcal{A}$ (*?t-of-g* (*gterm-to-None-Some u*))
by *auto blast*
have *depth* (*?t-og-g2 u*) = *depth* (*?t-of-g* (*gterm-to-None-Some u*))
proof (*induct u*)
case (*GFun f ts*) **then show** *?case*
by (*auto simp*: *comp-def intro*: *nth-args-depth-eqI*)
qed
from *this pigeonhole-tree-automata*[*OF - reach*] *card-st card-d* **obtain** *C2 C s v p* **where**

ctxt: *C2* $\neq \Box$ $C\langle s\rangle$ = *term-of-gterm* (*gterm-to-None-Some u*) *C2*$\langle v\rangle$ = *s* **and**
loop: *p* $|\in|$ *ta-der* $\mathcal{A}$ *v* $\wedge$ *p* $|\in|$ *ta-der* $\mathcal{A}$ *C2*$\langle$*Var p*$\rangle$ $\wedge$ *q* $|\in|$ *ta-der* $\mathcal{A}$ *C*$\langle$*Var p*$\rangle$
by *auto*
from *ctxt* **have** *gr*: *ground-ctxt C2 ground-ctxt C* **by** *auto* (*metis ground-ctxt-apply ground-term-of-gterm*)+
from *ta-der-to-ta-strict*[*OF - gr*(*1*)] *loop* **obtain** *q*′ **where**
to-strict: (*p* = *q*′ $\vee$ (*p*, *q*′) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$) $\wedge$ *p* $|\in|$ *ta-der-strict* $\mathcal{A}$ *C2*$\langle$*Var q*′$\rangle$
by *fastforce*
have $*$: $\exists$ *C*. *C2* = *map-funs-ctxt lift-None-Some C* $\wedge$ *C* $\neq \Box$ **using** *ctxt*(*1*, *2*)
apply (*auto simp flip*: *ctxt*(*3*))
by (*smt* (*verit*, *ccfv-threshold*) *map-ctxt.simps*(*1*) *map-funs-term-ctxt-decomp map-term-of-gterm*)
then have *q-p*: (*q*′, *p*) $\in$ *Q-inf* $\mathcal{A}$ **using** *to-strict ta-der-Q-inf*[*of p* $\mathcal{A}$ - *q*′] *ctxt*
by *auto*
then have *cycle*: (*q*′, *q*′) $\in$ *Q-inf* $\mathcal{A}$ **using** *to-strict* **by** (*auto intro*: *Q-inf-ta-eps-Q-inf*)
show *?thesis*
proof (*cases C* = $\Box$)
case *True* **then show** *?thesis*
using *cycle q-p loop* **by** (*auto intro*: *Q-inf-ta-eps-Q-inf*)
next
case *False*
obtain *q*′′ **where** *r*: *p* = *q*′′ $\vee$ (*p*, *q*′′) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ *q* $|\in|$ *ta-der-strict* $\mathcal{A}$ *C*$\langle$*Var q*′′$\rangle$
using *ta-der-to-ta-strict*[*OF conjunct2*[*OF conjunct2*[*OF loop*]] *gr*(*2*)] **by** *auto*
then have (*q*′′, *q*) $\in$ *Q-inf* $\mathcal{A}$ **using** *ta-der-Q-inf*[*of q* $\mathcal{A}$ - *q*′′] *ctxt False*
by *auto* (*metis* (*mono-tags*, *lifting*) *map-ctxt.simps*(*1*) *map-funs-term-ctxt-decomp map-term-of-gterm*)+
then show *?thesis* **using** *r*(*1*) *cycle q-p*
by (*auto simp*: *Q-inf-ta-eps-Q-inf intro*!: *exI*[*of* - *q*′])
(*meson Q-inf.trans Q-inf-ta-eps-Q-inf*)+
qed
qed

lemma *Q-infty-fQ-inf-e-conv*:
assumes *gta-lang Q* $\mathcal{A}$ $\subseteq$ {*gpair s t*| *s t*. *funas-gterm s* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$}
and $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-productive Q* $\mathcal{A}$
shows *Q-infty* $\mathcal{A}$ $\mathcal{F}$ = *fQ-inf-e* $\mathcal{A}$
using *Q-inf-impl-Q-inf-exec Q-inf-exec-impl-Q-inf*[*OF assms*]
by (*auto simp*: *fQ-inf-e.rep-eq*) *fastforce*

definition *Inf-reg-impl* **where**
Inf-reg-impl R = *Inf-reg R* (*fQ-inf-e* (*ta R*))

lemma *Inf-reg-impl-sound*:
assumes $\mathcal{L}$ $\mathcal{A}$ $\subseteq$ {*gpair s t*| *s t*. *funas-gterm s* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$}
and $\mathcal{Q}_r$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* (*ta* $\mathcal{A}$) **and** $\mathcal{Q}_r$ $\mathcal{A}$ $|\subseteq|$ *ta-productive* (*fin* $\mathcal{A}$) (*ta* $\mathcal{A}$)
shows $\mathcal{L}$ (*Inf-reg-impl* $\mathcal{A}$) = $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$))

using *Q-infty-fQ-inf-e-conv*[*of fin* $\mathcal{A}$ *ta* $\mathcal{A}$ $\mathcal{F}$] *assms*[*unfolded* $\mathcal{L}$*-def*]
by (*simp add*: *Inf-reg-impl-def*)

end
theory *Regular-Relation-Abstract-Impl*
imports *Pair-Automaton*
GTT-Transitive-Closure
RR2-Infinite-Q-infinity
Horn-Fset
begin

abbreviation *TA-of-lists* **where**
TA-of-lists Δ $\Delta_E \equiv$ *TA* (*fset-of-list* Δ) (*fset-of-list* Δ_E)

10 Computing the epsilon transitions for the composition of GTT's

definition Δ_ε*-rules* :: ('*q*, '*f*) *ta* ⇒ ('*q*, '*f*) *ta* ⇒ ('*q* × '*q*) *horn set* **where**
Δ_ε*-rules A B* =
{*zip ps qs* →$_h$ (*p*, *q*) |*f ps p qs q*. *f ps* → *p* |∈| *rules A* ∧ *f qs* → *q* |∈| *rules B* ∧ *length ps* = *length qs*} ∪
{[(*p*, *q*)] →$_h$ (*p*′, *q*) |*p p*′ *q*. (*p*, *p*′) |∈| *eps A*} ∪
{[(*p*, *q*)] →$_h$ (*p*, *q*′) |*p q q*′. (*q*, *q*′) |∈| *eps B*}

locale Δ_ε*-horn* =
fixes *A* :: ('*q*, '*f*) *ta* **and** *B* :: ('*q*, '*f*) *ta*
begin

sublocale *horn* Δ_ε*-rules A B* .

lemma Δ_ε*-infer0*:
infer0 = {(*p*, *q*) |*f p q*. *f* [] → *p* |∈| *rules A* ∧ *f* [] → *q* |∈| *rules B*}
unfolding *horn.infer0-def* Δ_ε*-rules-def*
using *zip-Nil*[*of* []]
by *auto* (*metis length-greater-0-conv zip-eq-Nil-iff*)+

lemma Δ_ε*-infer1*:
infer1 pq X = {(*p*, *q*) |*f ps p qs q*. *f ps* → *p* |∈| *rules A* ∧ *f qs* → *q* |∈| *rules B* ∧ *length ps* = *length qs* ∧
(*fst pq*, *snd pq*) ∈ *set* (*zip ps qs*) ∧ *set* (*zip ps qs*) ⊆ *insert pq X*} ∪
{(*p*′, *snd pq*) |*p p*′. (*p*, *p*′) |∈| *eps A* ∧ *p* = *fst pq*} ∪
{(*fst pq*, *q*′) |*q q*′. (*q*, *q*′) |∈| *eps B* ∧ *q* = *snd pq*}
unfolding Δ_ε*-rules-def horn-infer1-union*
apply (*intro arg-cong2*[*of* - - - - (∪)])
by (*auto simp*: *horn.infer1-def simp flip*: *ex-simps*(*1*)) *force*+

lemma Δ_ε*-sound*:
Δ_ε*-set A B* = *saturate*

```
proof (intro set-eqI iffI, goal-cases lr rl)
  case (lr x) obtain p q where x: x = (p, q) by (cases x)
  show ?case using lr unfolding x
  proof (induct)
    case (Δε-set-cong f ps p qs q) show ?case
      apply (intro infer[of zip ps qs (p, q)])
      subgoal using Δε-set-cong(1−3) by (auto simp: Δε-rules-def)
      subgoal using Δε-set-cong(3,5) by (auto simp: zip-nth-conv)
      done
  next
    case (Δε-set-eps1 p p' q) then show ?case
      by (intro infer[of [(p, q)] (p', q)]) (auto simp: Δε-rules-def)
  next
    case (Δε-set-eps2 q q' p) then show ?case
      by (intro infer[of [(p, q)] (p, q')]) (auto simp: Δε-rules-def)
  qed
next
  case (rl x) obtain p q where x: x = (p, q) by (cases x)
  show ?case using rl unfolding x
  proof (induct)
    case (infer as a) then show ?case
      using Δε-set-cong[of - map fst as fst a A map snd as snd a B]
        Δε-set-eps1[of - fst a A snd a B] Δε-set-eps2[of - snd a B fst a A]
      by (auto simp: Δε-rules-def)
  qed
qed

end
```

11 Computing the epsilon transitions for the transitive closure of GTT's

```
definition Δ-trancl-rules :: ('q, 'f) ta ⇒ ('q, 'f) ta ⇒ ('q × 'q) horn set where
  Δ-trancl-rules A B =
    Δε-rules A B ∪ {[(p, q), (q, r)] →h (p, r) |p q r. True}

locale Δ-trancl-horn =
  fixes A :: ('q, 'f) ta and B :: ('q, 'f) ta
begin

sublocale horn Δ-trancl-rules A B .

lemma Δ-trancl-infer0:
  infer0 = horn.infer0 (Δε-rules A B)
  by (auto simp: Δε-rules-def Δ-trancl-rules-def horn.infer0-def)

lemma Δ-trancl-infer1:
  infer1 pq X = horn.infer1 (Δε-rules A B) pq X ∪
```

```
   {(r, snd pq) |r p'. (r, p') ∈ X ∧ p' = fst pq} ∪
   {(fst pq, r) |q' r. (q', r) ∈ (insert pq X) ∧ q' = snd pq}
  unfolding Δ-trancl-rules-def horn-infer1-union Un-assoc
  apply (intro arg-cong2[of - - - - (∪)] HOL.refl)
  by (auto simp: horn.infer1-def simp flip: ex-simps(1)) force+

lemma Δ-trancl-sound:
  Δ-trancl-set A B = saturate
proof (intro set-eqI iffI, goal-cases lr rl)
  case (lr x) obtain p q where x: x = (p, q) by (cases x)
  show ?case using lr unfolding x
  proof (induct)
    case (Δ-set-cong f ps p qs q) show ?case
      apply (intro infer[of zip ps qs (p, q)])
     subgoal using Δ-set-cong(1−3) by (auto simp: Δ-trancl-rules-def Δ_ε-rules-def)
       subgoal using Δ-set-cong(3,5) by (auto simp: zip-nth-conv)
       done
  next
    case (Δ-set-eps1 p p' q) then show ?case
     by (intro infer[of [(p, q)] (p', q)]) (auto simp: Δ-trancl-rules-def Δ_ε-rules-def)
  next
    case (Δ-set-eps2 q q' p) then show ?case
     by (intro infer[of [(p, q)] (p, q')]) (auto simp: Δ-trancl-rules-def Δ_ε-rules-def)
  next
    case (Δ-set-trans p q r) then show ?case
    by (intro infer[of [(p,q), (q,r)] (p, r)]) (auto simp: Δ-trancl-rules-def Δ_ε-rules-def)
  qed
next
  case (rl x) obtain p q where x: x = (p, q) by (cases x)
  show ?case using rl unfolding x
  proof (induct)
    case (infer as a) then show ?case
      using Δ-set-cong[of - map fst as fst a A map snd as snd a B]
        Δ-set-eps1[of - fst a A snd a B] Δ-set-eps2[of - snd a B fst a A]
        Δ-set-trans[of fst a snd (hd as) A B snd a]
      by (auto simp: Δ-trancl-rules-def Δ_ε-rules-def)
  qed
qed

end
```

12 Computing the epsilon transitions for the transitive closure of pair automata

```
definition Δ-Atr-rules :: ('q × 'q) fset ⇒ ('q, 'f) ta ⇒ ('q, 'f) ta ⇒ ('q × 'q)
horn set where
  Δ-Atr-rules Q A B =
    {[] →_h (p, q) | p q. (p , q) |∈| Q} ∪
```

$\{[(p,\ q),(r,\ v)] \rightarrow_h (p,\ v)\ |p\ q\ r\ v.\ (q,\ r)\ |\in|\ \Delta_\varepsilon\ B\ A\}$

locale Δ-*Atr-horn* =
  fixes $Q :: ('q \times 'q)$ *fset* **and** $A :: ('q,\ 'f)$ *ta* **and** $B :: ('q,\ 'f)$ *ta*
begin

sublocale *horn* Δ-*Atr-rules* Q A B **.**

lemma Δ-*Atr-infer0*: *infer0* = *fset* Q
  by (*auto simp*: *horn.infer0-def* Δ-*Atr-rules-def*)

lemma Δ-*Atr-infer1*:
  infer1 pq X = $\{(p,\ snd\ pq)\ |\ p\ q.\ (p,\ q) \in X \wedge (q,\ fst\ pq)\ |\in|\ \Delta_\varepsilon\ B\ A\} \cup$
   $\{(fst\ pq,\ v)\ |\ q\ r\ v.\ (snd\ pq,\ r)\ |\in|\ \Delta_\varepsilon\ B\ A \wedge (r,\ v) \in X\} \cup$
   $\{(fst\ pq,\ snd\ pq)\ |\ q\ .\ (snd\ pq,\ fst\ pq)\ |\in|\ \Delta_\varepsilon\ B\ A\}$
  unfolding Δ-*Atr-rules-def horn-infer1-union*
  by (*auto simp*: *horn.infer1-def simp flip*: *ex-simps*(*1*)) *force*+

lemma Δ-*Atr-sound*:
  Δ-*Atrans-set* Q A B = *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
  case (*lr x*) **obtain** p q **where** x: $x = (p,\ q)$ **by** (*cases x*)
  show *?case* **using** *lr* **unfolding** x
  proof (*induct*)
    case (*base p q*)
    then show *?case*
      by (*intro infer*[*of* [] $(p,\ q)$]) (*auto simp*: Δ-*Atr-rules-def*)
  next
    case (*step p q r v*)
    then show *?case*
      by (*intro infer*[*of* $[(p,\ q),\ (r,\ v)]$ $(p,\ v)$]) (*auto simp*: Δ-*Atr-rules-def*)
  qed
next
  case (*rl x*) **obtain** p q **where** x: $x = (p,\ q)$ **by** (*cases x*)
  show *?case* **using** *rl* **unfolding** x
  proof (*induct*)
    case (*infer as a*) **then show** *?case*
      using *base*[*of fst a snd a Q A B*]
      using Δ-*Atrans-set.step*[*of fst a - Q A B snd a*]
      by (*auto simp*: Δ-*Atr-rules-def*) *blast*
  qed
qed

end

13 Computing the Q infinity set for the infinity predicate automaton

definition *Q-inf-rules* :: (*'q*, *'f option* × *'g option*) *ta* ⇒ (*'q* × *'q*) *horn set* **where**
Q-inf-rules A =
{[] →$_h$ (*ps* ! *i*, *p*) |*f ps p i*. (*None*, *Some f*) *ps* → *p* |∈| *rules A* ∧ *i* < *length ps*}
∪
{[(*p*, *q*)] →$_h$ (*p*, *r*) |*p q r*. (*q*, *r*) |∈| *eps A*} ∪
{[(*p*, *q*), (*q*, *r*)] →$_h$ (*p*, *r*) |*p q r*. *True*}

locale *Q-horn* =
fixes *A* :: (*'q*, *'f option* × *'g option*) *ta*
begin

sublocale *horn Q-inf-rules A* **.**

lemma *Q-infer0*:
infer0 = {(*ps* ! *i*, *p*) |*f ps p i*. (*None*, *Some f*) *ps* → *p* |∈| *rules A* ∧ *i* < *length ps*}
unfolding *horn.infer0-def Q-inf-rules-def* **by** *auto*

lemma *Q-infer1*:
infer1 pq X = {(*fst pq*, *r*) | *q r*. (*q*, *r*) |∈| *eps A* ∧ *q* = *snd pq*} ∪
{(*r*, *snd pq*) |*r p'*. (*r*, *p'*) ∈ *X* ∧ *p'* = *fst pq*} ∪
{(*fst pq*, *r*) |*q' r*. (*q'*, *r*) ∈ (*insert pq X*) ∧ *q'* = *snd pq*}
unfolding *Q-inf-rules-def horn-infer1-union*
by (*auto simp*: *horn.infer1-def simp flip*: *ex-simps*(*1*)) *force*+

lemma *Q-sound*:
Q-inf A = *saturate*
proof (*intro set-eqI iffI*, *goal-cases lr rl*)
case (*lr x*) **obtain** *p q* **where** *x*: *x* = (*p*, *q*) **by** (*cases x*)
show *?case* **using** *lr* **unfolding** *x*
proof (*induct*)
case (*trans p q r*)
then show *?case*
by (*intro infer*[*of* [(*p*,*q*), (*q*,*r*)] (*p*, *r*)])
(*auto simp*: *Q-inf-rules-def*)
next
case (*rule f qs q i*)
then show *?case*
by (*intro infer*[*of* [] (*qs* ! *i*, *q*)])
(*auto simp*: *Q-inf-rules-def*)
next
case (*eps p q r*)
then show *?case*
by (*intro infer*[*of* [(*p*, *q*)] (*p*, *r*)])
(*auto simp*: *Q-inf-rules-def*)
qed

next
case (*rl x*) **obtain** *p q* **where** *x*: *x* = (*p*, *q*) **by** (*cases x*)
show *?case* **using** *rl* **unfolding** *x*
proof (*induct*)
case (*infer as a*) **then show** *?case*
using *Q-inf.eps*[*of fst a - A snd a*]
using *Q-inf.trans*[*of fst a snd* (*hd as*) *A snd a*]
by (*auto simp*: *Q-inf-rules-def intro*: *Q-inf.rule*)
qed
qed

end

end
theory *Regular-Relation-Impl*
imports *Tree-Automata-Impl*
Regular-Relation-Abstract-Impl
Horn-Fset
begin

14 Computing the epsilon transitions for the composition of GTT's

definition Δ_ε*-infer0-cont* **where**
Δ_ε*-infer0-cont* Δ_A Δ_B =
(*let arules* = *filter* (λ *r*. *r-lhs-states r* = []) (*sorted-list-of-fset* Δ_A) *in*
let brules = *filter* (λ *r*. *r-lhs-states r* = []) (*sorted-list-of-fset* Δ_B) *in*
(*map* (*map-prod r-rhs r-rhs*) (*filter* (λ(*ra*, *rb*). *r-root ra* = *r-root rb*) (*List.product arules brules*))))

definition Δ_ε*-infer1-cont* **where**
Δ_ε*-infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
(*let* (*arules*, *aeps*) = (*sorted-list-of-fset* Δ_A, *sorted-list-of-fset* $\Delta_{A\varepsilon}$) *in*
let (*brules*, *beps*) = (*sorted-list-of-fset* Δ_B, *sorted-list-of-fset* $\Delta_{B\varepsilon}$) *in*
let prules = *List.product arules brules in*
(λ *pq bs*.
map (*map-prod r-rhs r-rhs*) (*filter* (λ(*ra*, *rb*). *case* (*ra*, *rb*) *of* (*TA-rule f ps p*, *TA-rule g qs q*) ⇒
f = *g* ∧ *length ps* = *length qs* ∧ (*fst pq*, *snd pq*) ∈ *set* (*zip ps qs*) ∧
set (*zip ps qs*) ⊆ *insert* (*fst pq*, *snd pq*) (*fset bs*)) *prules*) @
map (λ(*p*, *p'*). (*p'*, *snd pq*)) (*filter* (λ(*p*, *p'*) ⇒ *p* = *fst pq*) *aeps*) @
map (λ(*q*, *q'*). (*fst pq*, *q'*)) (*filter* (λ(*q*, *q'*) ⇒ *q* = *snd pq*) *beps*)))

locale Δ_ε*-fset* =
fixes Δ_A :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* **and** $\Delta_{A\varepsilon}$ :: ('*q* × '*q*) *fset*
and Δ_B :: ('*q*, '*f*) *ta-rule fset* **and** $\Delta_{B\varepsilon}$ :: ('*q* × '*q*) *fset*

begin

abbreviation A **where** $A \equiv TA\ \Delta_A\ \Delta_{A\varepsilon}$
abbreviation B **where** $B \equiv TA\ \Delta_B\ \Delta_{B\varepsilon}$

sublocale Δ_ε*-horn* A B **.**

sublocale *l*: *horn-fset* Δ_ε*-rules* A B Δ_ε*-infer0-cont* Δ_A Δ_B Δ_ε*-infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$
apply (*unfold-locales*)
unfolding Δ_ε*-horn.*Δ_ε*-infer0* Δ_ε*-horn.*Δ_ε*-infer1* Δ_ε*-infer0-cont-def* Δ_ε*-infer1-cont-def set-append Un-assoc*[*symmetric*]
unfolding *sorted-list-of-fset-simps union-fset*
subgoal
apply (*auto split*!: *prod.splits ta-rule.splits simp*: *comp-def fset-of-list-elem r-rhs-def*
map-prod-def fSigma.rep-eq image-def Bex-def)
apply (*metis ta-rule.exhaust-sel*)
done
unfolding *Let-def prod.case set-append Un-assoc*
apply (*intro arg-cong2*[*of* - - - - (∪)])
subgoal
apply (*auto split*!: *prod.splits ta-rule.splits*)
apply (*smt* (*verit*, *del-insts*) *Pair-inject map-prod-imageI mem-Collect-eq ta-rule.inject ta-rule.sel*(*3*))
done
by (*force simp add*: *image-def split*!: *prod.splits*)+

lemmas *infer* = *l.infer0 l.infer1*
lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition Δ_ε*-impl* **where**
Δ_ε*-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *horn-fset-impl.saturate-impl* (Δ_ε*-infer0-cont* Δ_A Δ_B) (Δ_ε*-infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)

lemma Δ_ε*-impl-sound*:
assumes Δ_ε*-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *Some xs*
shows *xs* = Δ_ε (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$)
using Δ_ε*-fset.saturate-impl-sound*[*OF assms*[*unfolded* Δ_ε*-impl-def*]]
unfolding Δ_ε*-horn.*Δ_ε*-sound*[*symmetric*]
by (*auto simp flip*: Δ_ε*.rep-eq*)

lemma Δ_ε*-impl-complete*:
fixes Δ_A :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* **and** Δ_B :: (*'q*, *'f*) *ta-rule fset*
and $\Delta_{\varepsilon A}$ :: (*'q* × *'q*) *fset* **and** $\Delta_{\varepsilon B}$ :: (*'q* × *'q*) *fset*

shows Δ_ε-*impl* Δ_A $\Delta_{\varepsilon A}$ Δ_B $\Delta_{\varepsilon B} \neq$ *None* **unfolding** Δ_ε-*impl-def*
by (*intro* Δ_ε-*fset.saturate-impl-complete*)
(*auto simp flip*: Δ_ε-*horn.*Δ_ε-*sound*)

lemma Δ_ε-*impl* [*code*]:
Δ_ε (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = *the* (Δ_ε-*impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)
using Δ_ε-*impl-complete*[*of* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$] Δ_ε-*impl-sound*[*of* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$]
by *force*

15 Computing the epsilon transitions for the transitive closure of GTT's

definition Δ-*trancl-infer0* **where**
Δ-*trancl-infer0* Δ_A Δ_B = Δ_ε-*infer0-cont* Δ_A Δ_B

definition Δ-*trancl-infer1* :: ('*q* :: *linorder* , '*f* :: *linorder*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset* ⇒ ('*q*, '*f*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset*
⇒ '*q* × '*q* ⇒ ('*q* × '*q*) *fset* ⇒ ('*q* × '*q*) *list* **where**
Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ *pq bs* =
Δ_ε-*infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ *pq bs* @
sorted-list-of-fset (
(λ(*r*, *p*'). (*r*, *snd pq*)) |‘| (*ffilter* (λ(*r*, *p*') ⇒ *p*' = *fst pq*) *bs*) |∪|
(λ(*q*', *r*). (*fst pq*, *r*)) |‘| (*ffilter* (λ(*q*', *r*) ⇒ *q*' = *snd pq*) (*finsert pq bs*)))

locale Δ-*trancl-list* =
fixes Δ_A :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* **and** $\Delta_{A\varepsilon}$ :: ('*q* × '*q*) *fset*
and Δ_B :: ('*q*, '*f*) *ta-rule fset* **and** $\Delta_{B\varepsilon}$ :: ('*q* × '*q*) *fset*
begin

abbreviation *A* **where** *A* ≡ *TA* Δ_A $\Delta_{A\varepsilon}$
abbreviation *B* **where** *B* ≡ *TA* Δ_B $\Delta_{B\varepsilon}$

sublocale Δ-*trancl-horn A B* **.**

sublocale *l*: *horn-fset* Δ-*trancl-rules A B*
Δ-*trancl-infer0* Δ_A Δ_B Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$
apply (*unfold-locales*)
unfolding Δ-*trancl-rules-def horn-infer0-union horn-infer1-union*
Δ-*trancl-infer0-def* Δ-*trancl-infer1-def* Δ_ε-*fset.infer set-append*
by (*auto simp flip*: *ex-simps*(*1*) *simp*: *horn.infer0-def horn.infer1-def intro*!: *arg-cong2*[*of* - - - - (∪)])

lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
horn-fset-impl.saturate-impl (Δ-*trancl-infer0* Δ_A Δ_B) (Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)

lemma Δ-*trancl-impl-sound*:
assumes Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *Some xs*
shows *xs* = Δ-*trancl* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$)
using Δ-*trancl-list.saturate-impl-sound*[*OF assms*[*unfolded* Δ-*trancl-impl-def*]]
unfolding Δ-*trancl-horn.*Δ-*trancl-sound*[*symmetric*] Δ-*trancl.rep-eq*[*symmetric*]
by *auto*

lemma Δ-*trancl-impl-complete*:
fixes Δ_A :: (′*q* :: *linorder*, ′*f* :: *linorder*) *ta-rule fset* **and** Δ_B :: (′*q*, ′*f*) *ta-rule fset*
and $\Delta_{A\varepsilon}$ :: (′*q* × ′*q*) *fset* **and** $\Delta_{B\varepsilon}$ :: (′*q* × ′*q*) *fset*
shows Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ ≠ *None*
unfolding Δ-*trancl-impl-def*
by (*intro* Δ-*trancl-list.saturate-impl-complete*)
(*auto simp flip*: Δ-*trancl-horn.*Δ-*trancl-sound*)

lemma Δ-*trancl-impl* [*code*]:
Δ-*trancl* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = (*the* (Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$))
using Δ-*trancl-impl-complete*[*of* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$]
using Δ-*trancl-impl-sound*[*of* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$]
by *force*

16 Computing the epsilon transitions for the transitive closure of pair automata

definition Δ-*Atr-infer1-cont* :: (′*q* :: *linorder* × ′*q*) *fset* ⇒ (′*q*, ′*f* :: *linorder*) *ta-rule fset* ⇒ (′*q* × ′*q*) *fset* ⇒
(′*q*, ′*f*) *ta-rule fset* ⇒ (′*q* × ′*q*) *fset* ⇒ ′*q* × ′*q* ⇒ (′*q* × ′*q*) *fset* ⇒ (′*q* × ′*q*) *list*
where
Δ-*Atr-infer1-cont Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
(*let G* = *sorted-list-of-fset* (*the* (Δ$_\varepsilon$-*impl* Δ_B $\Delta_{B\varepsilon}$ Δ_A $\Delta_{A\varepsilon}$)) *in*
(λ *pq bs*.
(*let bs-list* = *sorted-list-of-fset bs in*
map (λ (*p*, *q*). (*fst p*, *snd pq*)) (*filter* (λ (*p*, *q*). *snd p* = *fst q* ∧ *snd q* = *fst pq*) (*List.product bs-list G*)) @
map (λ (*p*, *q*). (*fst pq*, *snd q*)) (*filter* (λ (*p*, *q*). *snd p* = *fst q* ∧ *fst p* = *snd pq*) (*List.product G bs-list*)) @
map (λ (*p*, *q*). (*fst pq*, *snd pq*)) (*filter* (λ (*p*, *q*). *snd pq* = *p* ∧ *fst pq* = *q*) *G*))))

locale Δ-*Atr-fset* =
fixes *Q* :: (′*q* :: *linorder* × ′*q*) *fset* **and** Δ_A :: (′*q*, ′*f* :: *linorder*) *ta-rule fset* **and** $\Delta_{A\varepsilon}$ :: (′*q* × ′*q*) *fset*
and Δ_B :: (′*q*, ′*f*) *ta-rule fset* **and** $\Delta_{B\varepsilon}$ :: (′*q* × ′*q*) *fset*

begin

abbreviation A **where** $A \equiv TA\ \Delta_A\ \Delta_{A\varepsilon}$
abbreviation B **where** $B \equiv TA\ \Delta_B\ \Delta_{B\varepsilon}$

sublocale Δ-*Atr-horn* $Q\ A\ B$ **.**

lemma *infer1*:
  infer1 pq (*fset bs*) = *set* (Δ-*Atr-infer1-cont* $Q\ \Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$ *pq bs*)
proof −
  have {(*p*, *snd pq*) | *p q*. (*p*, *q*) ∈ (*fset bs*) ∧ (*q*, *fst pq*) |∈| Δ_ε *B A*} ∪
   {(*fst pq*, *v*) | *q r v*. (*snd pq*, *r*) |∈| Δ_ε *B A* ∧ (*r*, *v*) ∈ (*fset bs*)} ∪
   {(*fst pq*, *snd pq*) | *q* . (*snd pq*, *fst pq*) |∈| Δ_ε *B A*} = *set* (Δ-*Atr-infer1-cont* *Q* $\Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$ *pq bs*)
    unfolding Δ-*Atr-infer1-cont-def set-append Un-assoc*[*symmetric*] *Let-def*
    unfolding *sorted-list-of-fset-simps union-fset*
    apply (*intro arg-cong2*[*of* - - - - (∪)])
    apply (*simp-all add*: *fSigma-repeq flip*: Δ_ε-*impl fset-of-list-elem*)
    apply *force*+
    done
  then show *?thesis*
    using Δ-*Atr-horn.*Δ-*Atr-infer1*[*of Q A B pq fset bs*]
    by *simp*
qed

sublocale *l*: *horn-fset* Δ-*Atr-rules Q A B sorted-list-of-fset Q* Δ-*Atr-infer1-cont* *Q* $\Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$
  apply (*unfold-locales*)
  unfolding Δ-*Atr-horn.*Δ-*Atr-infer0 fset-of-list.rep-eq*
  using *infer1*
  by *auto*

lemmas *infer* = *l.infer0 l.infer1*
lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition Δ-*Atr-impl* $Q\ \Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$ =
  horn-fset-impl.saturate-impl (*sorted-list-of-fset Q*) (Δ-*Atr-infer1-cont* $Q\ \Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$)

lemma Δ-*Atr-impl-sound*:
  assumes Δ-*Atr-impl* $Q\ \Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$ = *Some xs*
  shows *xs* = Δ-*Atrans* *Q* (*TA* $\Delta_A\ \Delta_{A\varepsilon}$) (*TA* $\Delta_B\ \Delta_{B\varepsilon}$)
  using Δ-*Atr-fset.saturate-impl-sound*[*OF assms*[*unfolded* Δ-*Atr-impl-def*]]
  unfolding Δ-*Atr-horn.*Δ-*Atr-sound*[*symmetric*] Δ-*Atrans.rep-eq*[*symmetric*]
  by (*simp add*: *fset-inject*)

lemma Δ-*Atr-impl-complete*:
shows Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon} \neq$ *None* **unfolding** Δ-*Atr-impl-def*
by (*intro* Δ-*Atr-fset.saturate-impl-complete*)
(*auto simp*: *finite-*Δ-*Atrans-set simp flip*: Δ-*Atr-horn.*Δ-*Atr-sound*)

lemma Δ-*Atr-impl* [*code*]:
Δ-*Atrans Q* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = (*the* (Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$))
using Δ-*Atr-impl-complete*[*of Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$] Δ-*Atr-impl-sound*[*of Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$]
by *force*

17 Computing the Q infinity set for the infinity predicate automaton

definition *Q-infer0-cont* :: (′*q* :: *linorder*, ′*f* :: *linorder option* × ′*g* :: *linorder option*) *ta-rule fset* ⇒ (′*q* × ′*q*) *list* **where**
Q-infer0-cont Δ = *concat* (*sorted-list-of-fset* (
(λ *r*. *case r of TA-rule f ps p* ⇒ *map* (λ *x*. *Pair x p*) *ps*) |‘|
(*ffilter* (λ *r*. *case r of TA-rule f ps p* ⇒ *fst f* = *None* ∧ *snd f* ≠ *None* ∧ *ps* ≠ []) Δ)))

definition *Q-infer1-cont* :: (′*q* ::*linorder* × ′*q*) *fset* ⇒ ′*q* × ′*q* ⇒ (′*q* × ′*q*) *fset* ⇒ (′*q* × ′*q*) *list* **where**
Q-infer1-cont Δε =
(*let eps* = *sorted-list-of-fset* Δε *in*
(λ *pq bs*.
let bs-list = *sorted-list-of-fset bs in*
map (λ (*q*, *r*). (*fst pq*, *r*)) (*filter* (λ (*q*, *r*) ⇒ *q* = *snd pq*) *eps*) @
map (λ(*r*, *p*′). (*r*, *snd pq*)) (*filter* (λ(*r*, *p*′) ⇒ *p*′ = *fst pq*) *bs-list*) @
map (λ(*q*′, *r*). (*fst pq*, *r*)) (*filter* (λ(*q*′, *r*) ⇒ *q*′ = *snd pq*) (*pq* # *bs-list*))))

locale *Q-fset* =
fixes Δ :: (′*q* :: *linorder*, ′*f* :: *linorder option* × ′*g* :: *linorder option*) *ta-rule fset*
and Δε :: (′*q* × ′*q*) *fset*
begin

abbreviation *A* **where** *A* ≡ *TA* Δ Δε
sublocale *Q-horn A* **.**

sublocale *l*: *horn-fset Q-inf-rules A Q-infer0-cont* Δ *Q-infer1-cont* Δε
apply (*unfold-locales*)
unfolding *Q-horn.Q-infer0 Q-horn.Q-infer1 Q-infer0-cont-def Q-infer1-cont-def set-append Un-assoc*[*symmetric*]
unfolding *sorted-list-of-fset-simps union-fset*
subgoal
apply (*auto simp add*: *Bex-def split*!: *ta-rule.splits*)
apply (*rule-tac x* = *TA-rule* (*lift-None-Some f*) *ps p* **in** *exI*)

```
    apply (force dest: in-set-idx)+
    done
  unfolding Let-def set-append Un-assoc
  by (intro arg-cong2[of - - - - (∪)]) auto

lemmas saturate-impl-sound = l.saturate-impl-sound
lemmas saturate-impl-complete = l.saturate-impl-complete

end

definition Q-impl where
  Q-impl Δ Δε = horn-fset-impl.saturate-impl (Q-infer0-cont Δ) (Q-infer1-cont
Δε)

lemma Q-impl-sound:
  Q-impl Δ Δε = Some xs ⟹ fset xs = Q-inf (TA Δ Δε)
  using Q-fset.saturate-impl-sound unfolding Q-impl-def Q-horn.Q-sound .

lemma Q-impl-complete:
  Q-impl Δ Δε ≠ None
proof −
  let ?A = TA Δ Δε
  have ∗: Q-inf ?A ⊆ fset (𝒬 ?A |×| 𝒬 ?A)
    by (auto simp add: Q-inf-states-ta-states(1, 2) subrelI)
  have finite (Q-inf ?A)
    by (intro finite-subset[OF ∗]) simp
  then show ?thesis unfolding Q-impl-def
    by (intro Q-fset.saturate-impl-complete) (auto simp: Q-horn.Q-sound)
qed


definition Q-infinity-impl Δ Δε = (let Q = the (Q-impl Δ Δε) in
   snd |`| ((ffilter (λ (p, q). p = q) Q) |O| Q))

lemma Q-infinity-impl-fmember:
  q |∈| Q-infinity-impl Δ Δε ⟷ (∃ p. (p, p) |∈| the (Q-impl Δ Δε) ∧
    (p, q) |∈| the (Q-impl Δ Δε))
  unfolding Q-infinity-impl-def
  by (auto simp: Let-def image-iff Bex-def) fastforce

lemma loop-sound-correct [simp]:
  fset (Q-infinity-impl Δ Δε) = Q-inf-e (TA Δ Δε)
proof −
  obtain Q where [simp]: Q-impl Δ Δε = Some Q using Q-impl-complete[of Δ
Δε]
    by blast
  have fset Q = (Q-inf (TA Δ Δε))
    using Q-impl-sound[of Δ Δε]
    by (auto simp: fset-of-list.rep-eq)
```

then show *?thesis*
by (*force simp*: *Q-infinity-impl-fmember Let-def fset-of-list-elem fset-of-list.rep-eq*)
qed

lemma *fQ-inf-e-code* [*code*]:
fQ-inf-e (*TA* Δ Δε) = *Q-infinity-impl* Δ Δε
using *loop-sound-correct*
by (*auto simp add*: *fQ-inf-e.rep-eq*)

end

References

[1] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[2] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 242–248, 1990.

[3] G. Kucherov and M. Tajine. *Decidability of Regularity and Related Properties of Ground Normal Form Languages*, volume 118, pages 272–286. 01 2006.

[4] P. Lammich. Collections framework. *Archive of Formal Proofs*, Nov. 2009. https://isa-afp.org/entries/Collections.html, Formal proof development.

[5] P. Lammich. Tree automata. *Archive of Formal Proofs*, Nov. 2009. https://isa-afp.org/entries/Tree-Automata.html, Formal proof development.

[6] A. Lochmann, A. Middeldorp, F. Mitterwallner, and B. Felgenhauer. A verified decision procedure for the first-order theory of rewriting for linear variable-separated rewrite systems in Isabelle/HOL. In C. Hricu and A. Popescu, editors, *Proc. 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 250–263, 2021.