

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

Alexander Lochmann Bertram Felgenhauer
Christian Sternagel René Thiemann Thomas Sternagel

February 6, 2026

Abstract

Tree automata have good closure properties and therefore are 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 automata 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

1	Introduction	4
2	Preliminaries	4
2.1	Additional functionality on <i>Term.term</i> and <i>ctxt</i>	5
2.1.1	Type conversion	5
2.2	Properties on subterm at given position ($ $ -)	5
2.3	Properties on replace terms at a given position <i>replace-term-at</i>	5
2.4	Properties on <i>adapt-vars</i> and <i>adapt-vars-ctxt</i>	6
2.4.1	Equality on ground terms/contexts by positions and symbols	8
2.5	Misc	9
2.6	Ground constructions	15

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

2.6.1	Ground terms	15
2.6.2	Tree domains	21
2.6.3	Ground context	36
2.6.4	Multihole context closure	43
2.6.5	Signature closed property	45
2.6.6	Transitive closure preserves <i>all-ctxt-closed-gterm</i>	46
3	Tree automaton	54
3.1	Tree automaton definition and functionality	54
3.1.1	Reachability of a term induced by a tree automaton	55
3.1.2	Language acceptance	55
3.1.3	Trimming	56
3.1.4	Mapping over tree automata	57
3.1.5	Product construction (language intersection)	57
3.1.6	Union construction (language union)	57
3.1.7	Epsilon free and tree automaton accepting empty language	57
3.1.8	Relabeling tree automaton states to natural numbers	58
3.2	Powerset Construction for Tree Automata	102
3.3	Complement closure of regular languages	107
3.4	Pumping lemma	112
3.5	Myhill Nerode characterization for regular tree languages	119
4	Ground Tree Transducers (GTT)	121
4.1	(A)GTT reachable states	124
4.2	(A)GTT productive states	125
4.3	(A)GTT trimming	125
4.4	root-cleanliness	126
4.5	Relabeling	126
4.6	epsilon free GTTs	126
4.7	GTT closure under composition	127
4.8	GTT closure under transitivity	136
4.9	Pair automaton and anchored GTTs	142
4.10	Anchored gtt composition	152
4.11	Anchored gtt transitivity	153
4.12	Anchored gtt intersection	154
4.13	Anchored gtt trimming	154
5	Regular relations	155
5.1	Encoding pairs of terms	155
5.2	Decoding of pairs	157
5.3	Contexts to gpair	159
5.4	Encoding of lists of terms	161
5.5	RRn relations	163

5.6	Nullary automata	164
5.7	Pairing RR1 languages	164
5.8	Collapsing	174
5.9	Cylindrification	182
5.10	Projection	183
5.11	Permutation	187
5.12	Intersection	187
5.13	Difference	187
5.14	All terms over a signature	188
5.15	RR2 composition	189
6	Computing state derivation	203
7	Computing the restriction of tree automata to state set	204
8	Computing the epsilon transition for the product automaton	204
9	Computing reachability	205
9.1	Horn setup for reachable states	205
9.2	Computing productivity	206
9.2.1	Horn setup for productive states	207
9.3	Horn setup for power set construction states	208
9.4	Setup for the list implementation of reachable states	216
9.5	Setup for list implementation of productive states	218
9.6	Setup for the implementation of power set construction states	219
10	Computing the epsilon transitions for the composition of GTT's	234
11	Computing the epsilon transitions for the transitive closure of GTT's	235
12	Computing the epsilon transitions for the transitive closure of pair automata	236
13	Computing the Q infinity set for the infinity predicate automaton	237
14	Computing the epsilon transitions for the composition of GTT's	239
15	Computing the epsilon transitions for the transitive closure of GTT's	241

16 Computing the epsilon transitions for the transitive closure of pair automata	242
17 Computing the Q infinity set for the infinity predicate automaton	244

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.Subterm-and-Context
  First-Order-Terms.Term-More
  Polynomial-Factorization.Missing-List

```

begin

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

```
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
```

2.1.1 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-both f* ≡ *map-prod f f*

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

definition *adapt-vars-ctxt* :: ('f, 'q) ctxt ⇒ ('f, 'v) ctxt **where**
adapt-vars-ctxt = *map-vars-ctxt* (λ-. undefined)

2.2 Properties on subterm at given position (|-)

lemma *subst-at-ctxt-of-pos-term-eq-termD*:

assumes $s = t$ $p \in \text{poss } t$

shows $s \text{ |- } p = t \text{ |- } p \wedge \text{ctxt-of-pos-term } p \ s = \text{ctxt-of-pos-term } p \ t$ **using** *assms*

by *auto*

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

lemma *replace-term-at-not-poss* [*simp*]:

$p \notin \text{poss } s \implies s[p \leftarrow t] = s$

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*:

$p \in \text{poss } s \implies \text{replace-at } s \ p \ t = s[p \leftarrow t]$

by (induct s arbitrary: p) (auto simp: upd-conv-take-nth-drop)

lemma parallel-replace-term-commute [ac-simps]:

$p \perp q \implies s[p \leftarrow t][q \leftarrow u] = s[q \leftarrow u][p \leftarrow t]$

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 $i \neq j \implies (Fun f ts)[p \leftarrow t][q \leftarrow u] = (Fun f ts)[q \leftarrow u][p \leftarrow t]$

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

qed

lemma replace-term-at-above [simp]:

$p \leq_p q \implies s[q \leftarrow t][p \leftarrow u] = s[p \leftarrow u]$

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]:

$p <_p q \implies s[p \leftarrow t][q \leftarrow u] = s[p \leftarrow t][q -_p p \leftarrow u]$

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]: $C\langle s \rangle[\text{hole-pos } C \leftarrow t] = C\langle t \rangle$

by (induct C) auto

2.4 Properties on adapt-vars and adapt-vars-ctxt

lemma adapt-vars2:

$\text{adapt-vars} (\text{adapt-vars } t) = \text{adapt-vars } t$

by (induct t) (auto simp add: adapt-vars-def)

lemma adapt-vars-simps[code, simp]: $\text{adapt-vars} (Fun f ts) = Fun f (\text{map adapt-vars } ts)$

by (induct ts, auto simp: adapt-vars-def)

lemma adapt-vars-reverse: $\text{ground } t \implies \text{adapt-vars } t' = t \implies \text{adapt-vars } t = t'$

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 Hole = Hole
adapt-vars-ctxt (More f bef C aft) = More f (map adapt-vars bef) (adapt-vars-ctxt C) (map adapt-vars aft)
by (*simp-all add: adapt-vars-ctxt-def adapt-vars-def*)

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*)

2.4.1 Equality on ground terms/contexts by positions and symbols

lemma *fun-at-def'*:

fun-at t p = (if $p \in \text{poss } t$ then
 (case t |- p of *Var* $x \Rightarrow \text{Some } (\text{Inr } x) \mid \text{Fun } f \text{ ts} \Rightarrow \text{Some } (\text{Inl } f)$) else *None*)
 by (induct t p rule: *fun-at.induct*) *auto*

lemma *fun-at-None-nposs-iff*:

fun-at t p = *None* \longleftrightarrow $p \notin \text{poss } t$
 by (*auto simp: fun-at-def'*) (*meson term.case-eq-if*)

lemma *eq-term-by-poss-fun-at*:

assumes $\text{poss } s = \text{poss } t \wedge p. p \in \text{poss } s \implies \text{fun-at } s \text{ } p = \text{fun-at } t \text{ } p$
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** $\text{Fun}' = \text{this}$
show *?case*

proof (*cases t*)

case (*Var* x) **show** *?thesis* **using** $\text{Fun}'(3)$ [*of []*] **by** (*simp add: Var*)

next

case (*Fun* g ts)

have $*$: $\text{length } ss = \text{length } ts$

using $\text{Fun}'(3)$ *arg-cong*[*OF Fun'(2)*, *of* $\lambda P. \text{card } \{i \mid i \# p. i \# p \in P\}$]

by (*auto simp: Fun exI*[*of* $\lambda x. x \in \text{poss } -$, *OF empty-pos-in-poss*])

then have $i < \text{length } ss \implies \text{poss } (ss ! i) = \text{poss } (ts ! i)$ **for** i

using *arg-cong*[*OF Fun'(2)*, *of* $\lambda P. \{p. i \# p \in P\}$] **by** (*auto simp: Fun*)

then show *?thesis* **using** $*$ $\text{Fun}'(2)$ $\text{Fun}'(3)$ [*of []*] $\text{Fun}'(3)$ [*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 $p \in \text{poss } s \wedge p \in \text{poss } t$

and $\bigwedge q. \neg (p \leq_p q) \implies q \in \text{poss } s \longleftrightarrow q \in \text{poss } t$

and $(\bigwedge q. q \in \text{poss } s \implies \neg (p \leq_p q) \implies \text{fun-at } s \text{ } q = \text{fun-at } t \text{ } q)$

shows *ctxt-of-pos-term* p $s = \text{ctxt-of-pos-term } p \text{ } t$ **using** *assms*

proof (*induct p arbitrary: s t*)

case (*Cons* i p)

from $\text{Cons}(2, 3)$ $\text{Cons}(4, 5)$ [*of []*] **obtain** f ss ts **where** [*simp*]: $s = \text{Fun } f \text{ } ss \text{ } t$
 $= \text{Fun } f \text{ } ts$

by (*cases s; cases t auto*)

have *flt*: $j < i \implies j \# q \in \text{poss } s \implies \text{fun-at } s \text{ } (j \# q) = \text{fun-at } t \text{ } (j \# q)$ **for** $j \text{ } q$

by (*intro Cons(5) auto*)

have *fgt*: $i < j \implies j \# q \in \text{poss } s \implies \text{fun-at } s \text{ } (j \# q) = \text{fun-at } t \text{ } (j \# q)$ **for** j

```

q
  by (intro Cons(5)) auto
  have lt:  $j < i \implies j \# q \in \text{poss } s \longleftrightarrow j \# q \in \text{poss } t$  for  $j \ q$  by (intro Cons(4))
  auto
  have gt:  $i < j \implies j \# q \in \text{poss } s \longleftrightarrow j \# q \in \text{poss } t$  for  $j \ q$  by (intro Cons(4))
  auto
  from this[of - []] have  $i < j \implies j < \text{length } ss \longleftrightarrow j < \text{length } ts$  for  $j$  by auto
  from this Cons(2, 3) have  $l: \text{length } ss = \text{length } ts$  by auto (meson nat-neq-iff)
  have ctxt-of-pos-term  $p (ss ! i) = \text{ctxt-of-pos-term } p (ts ! i)$  using Cons(2, 3)
  Cons(4-)[of  $i \# q$  for  $q$ ]
  by (intro Cons(1)[of  $ss ! i \ ts ! i$ ]) auto
  moreover have  $\text{take } i \ ss = \text{take } i \ ts$  using  $l \ lt \ Cons(2, 3) \ fll$ 
  by (intro nth-equalityI) (auto intro!: eq-term-by-poss-fun-at)
  moreover have  $\text{drop } (Suc \ i) \ ss = \text{drop } (Suc \ i) \ ts$  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

```

2.5 Misc

```

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

```

```

lemma map-term-replace-at-dist:
   $p \in \text{poss } s \implies (\text{map-term } f \ g \ s)[p \leftarrow (\text{map-term } f \ g \ t)] = \text{map-term } f \ g \ (s[p \leftarrow t])$ 
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-Inl (Inl  $q$ )  $\longleftrightarrow$  True
| is-Inl (Inr  $q$ )  $\longleftrightarrow$  False

```

```

primrec is-Inr where
  is-Inr (Inr  $q$ )  $\longleftrightarrow$  True
| is-Inr (Inl  $q$ )  $\longleftrightarrow$  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 $n \leq \text{length } xs$ $ys = \text{take } n \text{ } xs @ x \# \text{drop } n \text{ } xs$

shows $ys \in \text{set } (\text{add-elem-list-lists } x \text{ } xs)$ **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'*:

$\text{set } (\text{add-elem-list-lists } x \text{ } xs) = \{ys \mid ys \text{ n. } n \leq \text{length } xs \wedge ys = \text{take } n \text{ } xs @ x \# \text{drop } n \text{ } xs\}$

using *add-elem-list-listsI add-elem-list-listsE*

by *fastforce*

fun *list-of-permutation-element-n* :: 'a \Rightarrow nat \Rightarrow 'a list \Rightarrow 'a list list **where**

list-of-permutation-element-n x 0 $L = []$

| *list-of-permutation-element-n* x (*Suc n*) $L = \text{concat } (\text{map } (\text{add-elem-list-lists } x) (\text{List.n-lists } n \text{ } L))$

lemma *list-of-permutation-element-n-conv*:

assumes $n \neq 0$

shows $\text{set } (\text{list-of-permutation-element-n } x \text{ } n \text{ } L) =$

$\{xs \mid xs \text{ i. } i < \text{length } xs \wedge (\forall j < \text{length } xs. j \neq i \longrightarrow xs ! j \in \text{set } L) \wedge \text{length } xs = n \wedge xs ! i = x\}$ (**is** *?Ls = ?Rs*)

proof (*intro equalityI*)

from *assms* **obtain** j **where** [*simp*]: $n = \text{Suc } j$ **using** *assms* **by** (*cases n*) *auto*

{fix ys **assume** $ys \in ?Ls$

then obtain xs i **where** *wit*: $xs \in \text{set } (\text{List.n-lists } j \text{ } L)$ $i \leq \text{length } xs$

$ys = \text{take } i \text{ } xs @ x \# \text{drop } i \text{ } xs$

by (*auto dest: add-elem-list-listsE*)

then have $i < \text{length } ys$ $\text{length } ys = \text{Suc } (\text{length } xs)$ $ys ! i = x$

by (*auto simp: nth-append*)

moreover have $\forall j < \text{length } ys. j \neq i \longrightarrow ys ! j \in \text{set } L$ **using** *wit(1, 2)*

by (*auto simp: wit(3) min-def nth-append set-n-lists*)

ultimately have $ys \in ?Rs$ **using** *wit(1) unfolding set-n-lists*

by *auto*}

then show $?Ls \subseteq ?Rs$ **by** *blast*

next

{fix xs **assume** $xs \in ?Rs$

then obtain i **where** *wit*: $i < \text{length } xs \wedge \forall j < \text{length } xs. j \neq i \longrightarrow xs ! j \in$

set L

$\text{length } xs = n$ $xs ! i = x$

by *blast*

then have $*$: $xs \in \text{set } (\text{add-elem-list-lists } (xs ! i) (\text{take } i \text{ } xs @ \text{drop } (\text{Suc } i) \text{ } xs))$

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]: $x \in \text{set } (\text{take } i \text{ } xs) \implies x \in \text{set } L$
 $x \in \text{set } (\text{drop } (\text{Suc } i) \text{ } xs) \implies x \in \text{set } L$ **for** x **using** wit(2)
by (auto simp: set-conv-nth)
have $xs \in ?Ls$ **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 $?Rs \subseteq ?Ls$ **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 \wedge ($\forall j < \text{length } xs. j \neq i \implies$
 $xs ! j \in \text{set } L$) \wedge length xs = n \wedge xs ! i = x})
proof (cases n)
case (Suc nat)
then have [simp]: $\text{Suc } \text{nat} \neq 0$ **by** auto
then show ?thesis
by (auto simp: list-of-permutation-element-n-conv)
qed auto

lemma list-of-permutation-element-n-conv':
assumes $x \in \text{set } L$ $0 < n$
shows set (list-of-permutation-element-n x n L) =
{xs. set xs \subseteq insert x (set L) \wedge length xs = n \wedge $x \in \text{set } xs$ }
proof -
from assms(2) **have** *: $n \neq 0$ **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:
 $x \in \text{set } xs \implies \exists i < \text{length } xs. xs ! i = x$
by (induct xs) force+

lemma set-list-subset-eq-nth-conv:
set xs $\subseteq A \iff (\forall i < \text{length } xs. xs ! i \in A)$
by (metis in-set-conv-nth subset-code(1))

lemma map-eq-nth-conv:
 $\text{map } f \text{ } xs = \text{map } g \text{ } ys \iff \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } ys. f (xs ! i) =$
 $g (ys ! i))$
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 $i < \text{length } xs$ then $xs ! i$ else if $i = \text{length } xs$ then y else $zs ! (i - \text{Suc } (\text{length } xs))$)
by (*cases i length xs rule: linorder-cases, auto simp: nth-append*)

lemma *map-prod-times*:
 $f ' A \times g ' B = \text{map-prod } f g ' (A \times B)$
by *auto*

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

lemma *trancl-map*:
assumes *simu*: $\bigwedge x y. (x, y) \in r \implies (f x, f y) \in s$
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*:
 $\text{map-both } f ' R^+ \subseteq (\text{map-both } f ' R)^+$
proof –
have $(f x, f y) \in (\text{map-both } f ' R)^+$ **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 $(\text{map-both } f ' \text{Restr } R X)^+ = \text{map-both } f ' (\text{Restr } R X)^+$
proof –
have [*simp*]:
 $\text{map-prod } (\text{inv-into } X f \circ f) (\text{inv-into } X f \circ f) ' \text{Restr } R X = \text{Restr } R X$
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*]:
 $\text{map-prod } (f \circ \text{inv-into } X f) (f \circ \text{inv-into } X f) ' (\text{map-both } f ' \text{Restr } R X)^+ =$
 $(\text{map-both } f ' \text{Restr } R X)^+$
using *f-inv-into-f[of - f X] subsetD[OF trancl-mono-subset[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]*
 $\text{image-mono}[OF \text{trancl-map-prod-mono}[of \text{inv-into } X f \text{map-both } f ' \text{Restr } R X], \text{of } \text{map-both } f]$

by (*intro equalityI*) (*simp-all add: image-comp map-prod.comp*)
qed

lemma *inj-on-trancl-map-both*:

assumes *inj-on* f (*fst* ' $R \cup \text{snd}$ ' R)
shows $(\text{map-both } f \text{ ' } R)^+ = \text{map-both } f \text{ ' } R^+$

proof –

have [*simp*]: $\text{Restr } R$ (*fst* ' $R \cup \text{snd}$ ' R) = R
by (*force simp: image-def*)

then show *?thesis* **using** *assms*

using *trancl-map-both-Restr*[*of f fst ' R \cup \text{snd} ' R R*]

by *simp*

qed

lemma *kleene-induct*:

$A \subseteq X \implies B \circ X \subseteq X \implies X \circ C \subseteq X \implies B^* \circ A \circ C^* \subseteq X$

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*:

$A \subseteq X \implies B \circ X \subseteq X \implies X \circ C \subseteq X \implies B^+ \circ A \circ C^+ \subseteq X$

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*:

$B \circ A = \{\} \implies (A \cup B)^* = A^* \cup A^* \circ B^*$

by (*metis R-O-Id empty-subsetI relcomp-distrib rtrancl-U-push rtrancl-reflcl-absorb sup-commute*)

lemma *trancl-Un2-separatorE*:

assumes $B \circ A = \{\}$

shows $(A \cup B)^+ = A^+ \cup A^+ \circ B^+ \cup B^+$ (**is** *?Ls = ?Rs*)

proof –

{**fix** $x y$ **assume** $(x, y) \in ?Ls$

then have $(x, y) \in ?Rs$ **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** $q = \text{Inr } p$
using *assms* **by** (*cases q*) *auto*

lemma *is-InlE*:
assumes *is-Inl q*
obtains *p* **where** $q = \text{Inl } p$
using *assms* **by** (*cases q*) *auto*

lemma *not-is-Inr-is-Inl [simp]*:
 $\neg \text{is-Inl } t \longleftrightarrow \text{is-Inr } t$
 $\neg \text{is-Inr } t \longleftrightarrow \text{is-Inl } t$
by (*cases t, auto*)**+**

lemma [*simp*]: $\text{remove-sum} \circ \text{Inl} = \text{id}$ **by** *auto*

abbreviation *CInl* :: $'q \Rightarrow 'q + 'q$ **where** $CInl \equiv \text{Inl}$
abbreviation *CInr* :: $'q \Rightarrow 'q + 'q$ **where** $CInr \equiv \text{Inr}$

lemma *inj-CInl*: $\text{inj } CInl \text{ inj } CInr$ **using** *inj-Inl inj-Inr* **by** *blast+*

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

end

2.6 Ground constructions

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

2.6.1 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 $'f \ \text{gterm} =$
 $\text{GFun } (\text{groot-sym}: 'f) (\text{gargs}: 'f \ \text{gterm list})$

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

fun *term-of-gterm* **where**

$term\text{-of-gterm} (GFun\ f\ ts) = Fun\ f\ (map\ term\text{-of-gterm}\ ts)$

fun *gterm-of-term* **where**
 $gterm\text{-of-term} (Fun\ f\ ts) = GFun\ f\ (map\ gterm\text{-of-term}\ ts)$

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

lemma *groot-sym-groot-conv*:
 $groot\text{-sym}\ t = fst\ (groot\ t)$
by (*cases t*) *auto*

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

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

lemma *ground-term-of-gterm* [*simp*]:
 $ground\ (term\text{-of-gterm}\ s)$
by (*induct s*) *auto*

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

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

lemma *inj-term-of-gterm*:
 $inj\text{-on}\ term\text{-of-gterm}\ X$
by (*metis inj-on-def term-of-gterm-inv*)

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

lemma *ground-term-to-gtermD*:
 $ground\ t \implies \exists t'. t = term\text{-of-gterm}\ t'$
by (*metis gterm-of-term-inv*)

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

lemma *map-gterm-of-term* [*simp*]:

```

ground t  $\implies$  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  $\in$  gposs t  $\implies$  p  $\in$  poss (term-of-gterm t)
  by (simp add: poss-gposs-mem-conv)

fun gfun-at :: 'f gterm  $\Rightarrow$  pos  $\Rightarrow$  '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 $exInl \equiv case\text{-}sum (\lambda x. x) (\lambda _ . undefined)$

lemma $gfun\text{-}at\text{-}gterm\text{-}of\text{-}term$ [simp]:

$ground\ s \implies map\text{-}option\ exInl (fun\text{-}at\ s\ p) = gfun\text{-}at (gterm\text{-}of\text{-}term\ s)\ p$

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 $gfun\text{-}at\text{-}gterm\text{-}of\text{-}term'$ [simp] = $gfun\text{-}at\text{-}gterm\text{-}of\text{-}term$ [OF ground-term-of-gterm, unfolded term-of-gterm-inv]

lemma $gfun\text{-}at\text{-}None\text{-}ngposs\text{-}iff$: $gfun\text{-}at\ s\ p = None \longleftrightarrow p \notin gposs\ s$

by (induct rule: gfun-at.induct) auto

lemma $gfun\text{-}at\text{-}map\text{-}gterm$ [simp]:

$gfun\text{-}at (map\text{-}gterm\ f\ t)\ p = map\text{-}option\ f (gfun\text{-}at\ t\ p)$

by (induct t arbitrary: p; case-tac p) (auto simp: comp-def)

lemma $set\text{-}gterm\text{-}gposs\text{-}conv$:

$set\text{-}gterm\ t = \{the (gfun\text{-}at\ t\ p) \mid p. p \in gposs\ t\}$

proof (induct t)

case (GFun f ts)

note [simp] = $gfun\text{-}at\text{-}gterm\text{-}of\text{-}term$ [OF ground-term-of-gterm, unfolded term-of-gterm-inv]

have [simp]: $\{the (map\text{-}option\ exInl (fun\text{-}at (Fun\ f (map\ term\text{-}of\text{-}gterm\ ts :: (-, unit)\ term\ list))\ p)) \mid p.$

$\exists i\ pa. p = i \# pa \wedge i < length\ ts \wedge pa \in gposs (ts ! i)\} =$

$(\bigcup x \in \{ts ! i \mid i. i < length\ ts\}. \{the (gfun\text{-}at\ x\ p) \mid p. p \in gposs\ x\})$

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]: $(\bigcup x \in \{ts ! i \mid i. i < length\ ts\}. \{the (gfun\text{-}at\ x\ p) \mid p. p \in gposs\ x\})$

=

$\{the (gfun\text{-}at (GFun\ f\ ts)\ p) \mid p. \exists i\ pa. p = i \# pa \wedge i < length\ ts \wedge pa \in gposs (ts ! i)\}$

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 \longleftrightarrow 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 \longleftrightarrow p \notin gposs (ts ! i)*

using *fun-at-None-nposs-iff* **by** (*metis poss-gposs-mem-conv*)

have [*simp*]: *None = gfun-at (ts ! i) p \longleftrightarrow p \notin 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 \Rightarrow pos \Rightarrow '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 \in 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 \in 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-at s p) = {x | 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 $gfun\text{-}at\ (gsubst\text{-}at\ s\ p)\ q = gfun\text{-}at\ s\ (p\ @\ q)$
using *assms* **by** (*induct s arbitrary: p q*) *auto*

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

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

Replace terms at position

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

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

lemma *parallel-replace-gterm-commute* [*ac-simps*]:
 $p \perp q \implies s[p \leftarrow t]_G[q \leftarrow u]_G = s[q \leftarrow u]_G[p \leftarrow t]_G$
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 $i \neq j \implies (GFun\ f\ ts)[p \leftarrow t]_G[q \leftarrow u]_G = (GFun\ f\ ts)[q \leftarrow u]_G[p \leftarrow t]_G$
by (*auto simp: list-update-swap*)
then show $?case$ **using** $GFun(1)[OF\ nth\text{-}mem, of\ j\ ps\ qs]$ $GFun(2)$
by (*cases i = j*) *auto*
qed

lemma *replace-gterm-at-above* [*simp*]:
 $p \leq_p q \implies s[q \leftarrow t]_G[p \leftarrow u]_G = s[p \leftarrow u]_G$
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]:
 $p <_p q \implies s[p \leftarrow t]_G[q \leftarrow u]_G = s[p \leftarrow t[q \leftarrow_p p \leftarrow u]_G]_G$
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]:
 $p \neq [] \implies \text{groot-sym } s[p \leftarrow t]_G = \text{groot-sym } s$
 by (cases s; cases p) auto

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

lemma *replace-gterm-conv*:
 $p \in \text{gposs } s \implies (\text{term-of-gterm } s)[p \leftarrow (\text{term-of-gterm } t)] = \text{term-of-gterm } (s[p \leftarrow t]_G)$
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

2.6.2 Tree domains

type-synonym *gdomain* = unit *gterm*

abbreviation *gdomain* where
 $\text{gdomain} \equiv \text{map-gterm } (\lambda \cdot. ())$

lemma *gdomain-id*:
 $\text{gdomain } t = t$
proof –
 have [simp]: $(\lambda \cdot. ()) = \text{id}$ by auto
 then show ?thesis by (simp add: *gterm.map-id*)
qed

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

Union of tree domains

fun *gunion* :: $\text{gdomain} \Rightarrow \text{gdomain} \Rightarrow \text{gdomain}$ where

$gunion (GFun f ss) (GFun g ts) = GFun () (map (\lambda i.$
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]:

$gposs (gunion s t) = gposs s \cup gposs t$

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 $gsubt-at (gunion s t) p = gsubt-at s p$

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 $gsubt-at (gunion s t) p = gsubt-at t p$

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 $gunion (gsubt-at s p) (gsubt-at t p) = gsubt-at (gunion s t) p$

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 $gfun-at t p = (if p \in gposs t then Some () else None)$

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 gposs-gmem-conv*)

definition *gunions* :: *gdomain list* \Rightarrow *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) = $\{\}\cup\cup\{gposs\ t \mid t \in set\ ts\}$
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* \Rightarrow 'f) \Rightarrow *gdomain* \Rightarrow 'f *gterm* **where**

glabel h (GFun f ts) = *GFun (h []) (map (\lambda i. glabel (h o (#) i) (ts ! i)) [0..*length ts*])*

end

lemma *map-gterm-glabel*:

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

lemma *gfun-at-glabel* [*simp*]:

gfun-at (glabel f t) p = (*if p* \in *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 o gfun-at t) (gdomain t) = *map-gterm (f o Some) t*
by (*induct t*) (*auto simp: comp-def intro!: nth-equalityI*)

lemma *gfun-at-nongposs* [*simp*]:

p \notin *gposs t* \Longrightarrow *gfun-at t p* = *None*
using *gfun-at-glabel*[*of the* \circ *gfun-at t gdomain t p*, *unfolded glabel-map-gterm-conv*]
by (*simp add: comp-def option.map-ident*)

lemma *gfun-at-poss*:

p \in *gposs t* \Longrightarrow $\exists f.$ *gfun-at t p* = *Some f*
using *gfun-at-glabel*[*of the* \circ *gfun-at t gdomain t p*, *unfolded glabel-map-gterm-conv*]
by (*auto simp: comp-def*)

lemma *gfun-at-possE*:

assumes $p \in gposs\ t$

obtains f **where** $gfun\text{-at}\ t\ p = Some\ f$

using *assms gfun-at-poss* **by** *blast*

lemma *gfun-at-poss-gpossD*:

$gfun\text{-at}\ t\ p = Some\ f \implies p \in gposs\ t$

by (*metis gfun-at-nongposs option.distinct(1)*)

function symbols of a ground term

primrec *funas-gterm* :: ' $f\ gterm \Rightarrow (f \times nat)$ set **where**

$funas\text{-gterm}\ (GFun\ f\ ts) = \{(f, length\ ts)\} \cup \bigcup (set\ (map\ funas\text{-gterm}\ ts))$

lemma *funas-gterm-gterm-of-term*:

$ground\ t \implies funas\text{-gterm}\ (gterm\text{-of-term}\ t) = funas\text{-term}\ t$

by (*induct t*) (*auto simp: funas-gterm-def*)

lemma *funas-term-of-gterm-conv*:

$funas\text{-term}\ (term\text{-of-gterm}\ t) = funas\text{-gterm}\ t$

by (*induct t*) (*auto simp: funas-gterm-def*)

lemma *funas-gterm-map-gterm*:

assumes $funas\text{-gterm}\ t \subseteq \mathcal{F}$

shows $funas\text{-gterm}\ (map\text{-gterm}\ f\ t) \subseteq (\lambda\ (h, n). (f\ h, n))\ ` \mathcal{F}$

using *assms* **by** (*induct t*) (*auto simp: funas-gterm-def*)

lemma *gterm-of-term-inj*:

assumes $\bigwedge t. t \in S \implies ground\ t$

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 $p \in gposs\ s$

shows $funas\text{-gterm}\ (gsubt\text{-at}\ s\ p) \subseteq funas\text{-gterm}\ s$ **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**

$gterms\ \mathcal{F} \equiv \{s. funas\text{-gterm}\ s \subseteq \mathcal{F}\}$

lemma *gterms-mono*:

$\mathcal{G} \subseteq \mathcal{F} \implies gterms\ \mathcal{G} \subseteq gterms\ \mathcal{F}$

by *auto*

inductive-set \mathcal{T}_G **for** \mathcal{F} **where**

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

lemma \mathcal{T}_G -sound:

$s \in \mathcal{T}_G\ \mathcal{F} \implies funas-gterm\ s \subseteq \mathcal{F}$

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:

$funas-gterm\ s \subseteq \mathcal{F} \implies s \in \mathcal{T}_G\ \mathcal{F}$

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]:

$s \in \mathcal{T}_G\ \mathcal{F} \implies s \in \mathcal{T}_G\ \mathcal{G} \implies s \in \mathcal{T}_G\ (\mathcal{F} \cap \mathcal{G})$

by (auto simp: \mathcal{T}_G -funas-gterm-conv \mathcal{T}_G -equivalent-def)

lemma \mathcal{T}_G -mono:

$\mathcal{G} \subseteq \mathcal{F} \implies \mathcal{T}_G\ \mathcal{G} \subseteq \mathcal{T}_G\ \mathcal{F}$

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

$funas-grel\ \mathcal{R} = \bigcup ((\lambda (s, t). funas-gterm\ s \cup funas-gterm\ t) \text{ ` } \mathcal{R})$

end

theory FSet-Utills

imports HOL-Library.FSet

HOL-Library.List-Lexorder

Ground-Terms

begin

context

includes fset.lifting

begin

lift-definition $fCollect :: ('a \Rightarrow bool) \Rightarrow 'a \text{ fset} \text{ is } \lambda P. \text{ if finite (Collect P) then Collect P else \{\}$

by *auto*

lift-definition $fSigma :: 'a \text{ fset} \Rightarrow ('a \Rightarrow 'b \text{ fset}) \Rightarrow ('a \times 'b) \text{ fset} \text{ is Sigma}$

by *auto*

lift-definition $is\text{-empty} :: 'a \text{ fset} \Rightarrow bool \text{ is Set.is-empty} .$

lift-definition $fremove :: 'a \Rightarrow 'a \text{ fset} \Rightarrow 'a \text{ fset} \text{ is Set.remove}$

by *simp*

lift-definition $finj\text{-on} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ fset} \Rightarrow bool \text{ is inj-on} .$

lift-definition $the\text{-finv-into} :: 'a \text{ fset} \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ is the-inv-into} .$

lemma $fCollect\text{-memberI}$ [*intro!*]:

$finite (Collect P) \Longrightarrow P x \Longrightarrow x \in fCollect P$

by *transfer auto*

lemma $fCollect\text{-member}$ [*iff*]:

$x \in fCollect P \longleftrightarrow finite (Collect P) \wedge P x$

by *transfer (auto split: if-splits)*

lemma $fCollect\text{-cong}$: $(\bigwedge x. P x = Q x) \Longrightarrow fCollect P = fCollect Q$

by *presburger*

end

syntax

$\text{-fColl} :: \text{pttrn} \Rightarrow bool \Rightarrow 'a \text{ set} \quad (\langle (1\{|-./ -\}) \rangle)$

syntax-consts

$\text{-fColl} \Rightarrow fCollect$

translations

$\{x. P\} \Rightarrow CONST fCollect (\lambda x. P)$

syntax (*ASCII*)

$\text{-fCollect} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow 'a \text{ set} \quad (\langle (1\{(-/|:| -)/ -\}) \rangle)$

syntax

$\text{-fCollect} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow bool \Rightarrow 'a \text{ set} \quad (\langle (1\{(-/|\in| -)/ -\}) \rangle)$

syntax-consts

$\text{-fCollect} \Rightarrow fCollect$

translations

$\{p|:|A. P\} \rightarrow CONST fCollect (\lambda p. p \in A \wedge P)$

syntax

$\text{-fSetcompr} :: 'a \Rightarrow \text{idts} \Rightarrow bool \Rightarrow 'a \text{ fset} \quad (\langle (1\{|-|./ -\}) \rangle)$

syntax-consts

$\text{-fSetcompr} \Rightarrow fCollect$

parse-translation \langle

```

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 const-syntax ⟨-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 const-syntax ⟨-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 const-syntax ⟨-fCollect⟩ $ x $ A $ P else
            M
          | - => M
        end
      end;
end;

```

in [(**const-syntax** $\langle fCollect \rangle$, $setcompr-tr'$)] end
 \rangle

syntax

$-fSigma :: pptrn \Rightarrow 'a fset \Rightarrow 'b fset \Rightarrow ('a \times 'b) set \ (\langle (3fSIGMA \ -|:-./ \ -) \rangle [0, 0, 10] 10)$

syntax-consts

$-fSigma \equiv fSigma$

translations

$fSIGMA \ x|:|A. B \equiv CONST \ fSigma \ A \ (\lambda x. B)$

notation

$ffUnion \ (\langle | \cup | \rangle)$

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 $fCollectD$: $a \in | \{ | x . P \ x \} \implies P \ a$

by $transfer \ (auto \ split: \ if-splits)$

lemma $fCollectI$: $P \ a \implies finite \ (Collect \ P) \implies a \in | \{ | x . P \ x \}$

by ($auto \ intro: \ fCollect-memberI$)

lemma $fCollect-empty-eq$ [simp]: $fCollect \ P = \{ | \} \longleftrightarrow (\forall x. \neg P \ x) \vee infinite$

(Collect P)
 by auto

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

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

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

lemma *ffilter-simp*: $ffilter\ P\ A = \{a \mid a \in A. P\ a\}$
 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 $A \mid \times \mid B \equiv fSigma\ A\ (\lambda\cdot. B)$

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 $fSigma\text{-Int-distrib1} = Sigma\text{-Int-distrib1}[Transfer.transferred]$
lemmas $fSigma\text{-Int-distrib2} = Sigma\text{-Int-distrib2}[Transfer.transferred]$
lemmas $fSigma\text{-Diff-distrib1} = Sigma\text{-Diff-distrib1}[Transfer.transferred]$
lemmas $fSigma\text{-Diff-distrib2} = Sigma\text{-Diff-distrib2}[Transfer.transferred]$
lemmas $fSigma\text{-Union} = Sigma\text{-Union}[Transfer.transferred]$
lemmas $fTimes\text{-Un-distrib1} = Times\text{-Un-distrib1}[Transfer.transferred]$
lemmas $fTimes\text{-Int-distrib1} = Times\text{-Int-distrib1}[Transfer.transferred]$
lemmas $fTimes\text{-Diff-distrib1} = Times\text{-Diff-distrib1}[Transfer.transferred]$
lemmas $fTimes\text{-empty} [simp] = Times\text{-empty}[Transfer.transferred]$
lemmas $ftimes\text{-subset-iff} = times\text{-subset-iff}[Transfer.transferred]$
lemmas $ftimes\text{-eq-iff} = times\text{-eq-iff}[Transfer.transferred]$
lemmas $fst\text{-image-times} [simp] = fst\text{-image-times}[Transfer.transferred]$
lemmas $snd\text{-image-times} [simp] = snd\text{-image-times}[Transfer.transferred]$
lemmas $snd\text{-image-Sigma} = snd\text{-image-Sigma}[Transfer.transferred]$
lemmas $insert\text{-Times-insert} = insert\text{-Times-insert}[Transfer.transferred]$
lemmas $fTimes\text{-Int-Times} = Times\text{-Int-Times}[Transfer.transferred]$
lemmas $image\text{-paired-Times} = image\text{-paired-Times}[Transfer.transferred]$
lemmas $fproduct\text{-swap} = product\text{-swap}[Transfer.transferred]$
lemmas $fswap\text{-product} = swap\text{-product}[Transfer.transferred]$
lemmas $fsubset\text{-fst-snd} = subset\text{-fst-snd}[Transfer.transferred]$
lemmas $map\text{-prod-ftimes} = map\text{-prod-times}[Transfer.transferred]$

lemma $fCollect\text{-case-prod} [simp]:$
 $\{|(a, b). P a \wedge Q b|\} = fCollect P \times fCollect Q$
by $transfer (auto dest: finite\text{-cartesian-productD1 finite\text{-cartesian-productD2})$
lemma $fCollect\text{-case-prodD}:$
 $x \in \{|(x, y). A x y|\} \implies A (fst x) (snd x)$
by $auto$

lemmas $fCollect\text{-case-prod-Sigma} = Collect\text{-case-prod-Sigma}[Transfer.transferred]$
lemmas $fst\text{-image-Sigma} = fst\text{-image-Sigma}[Transfer.transferred]$
lemmas $image\text{-split-eq-Sigma} = image\text{-split-eq-Sigma}[Transfer.transferred]$

— Dealing with transitive closure

lift-definition $ftrancl :: ('a \times 'a) fset \Rightarrow ('a \times 'a) fset \langle (-|+)\rangle [1000] 999$ **is**
 $trancl$
by $auto$

lemmas $fr\text{-into-trancl} [intro, Pure.intro] = r\text{-into-trancl}[Transfer.transferred]$
lemmas $ftrancl\text{-into-trancl} [Pure.intro] = trancl\text{-into-trancl}[Transfer.transferred]$
lemmas $ftrancl\text{-induct} [consumes 1, case\text{-names Base Step}] = trancl\text{-induct}[Transfer.transferred]$
lemmas $ftrancl\text{-mono} = trancl\text{-mono}[Transfer.transferred]$
lemmas $ftrancl\text{-trans} [trans] = trancl\text{-trans}[Transfer.transferred]$
lemmas $ftrancl\text{-empty} [simp] = trancl\text{-empty}[Transfer.transferred]$

lemmas $ftranclE$ [cases set: $ftrancl$] = $tranclE$ [Transfer.transferred]
lemmas $converse-ftrancl-induct$ [consumes 1, case-names Base Step] = $converse-trancl-induct$ [Transfer.transferred]
lemmas $converse-ftranclE$ = $converse-tranclE$ [Transfer.transferred]
lemma $in-ftrancl-UnI$:
 $x \in R^+ \vee x \in S^+ \implies x \in (R \cup S)^+$
by $transfer$ (auto simp add: trancl-mono)

lemma $ftranclD$:
 $(x, y) \in R^+ \implies \exists z. (x, z) \in R \wedge (z = y \vee (z, y) \in R^+)$
by (induct rule: $ftrancl-induct$) (auto, meson $ftrancl-into-trancl$)

lemma $ftranclD2$:
 $(x, y) \in R^+ \implies \exists z. (x = z \vee (x, z) \in R^+) \wedge (z, y) \in R$
by (induct rule: $ftrancl-induct$) auto

lemma $not-ftrancl-into$:
 $(x, z) \notin r^+ \implies (y, z) \in r \implies (x, y) \notin r^+$
by $transfer$ (auto simp add: trancl.trancl-into-trancl)

lemmas $ftrancl-map-both-fRestr$ = $trancl-map-both-Restr$ [Transfer.transferred]
lemma $ftrancl-map-both-fsubset$:
 $finj-on f X \implies R \subseteq X \times X \implies (map-both f \upharpoonright R)^+ = map-both f \upharpoonright R^+$
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 $upt-fset$:
 $fset-of-list [i..<j] = fCollect (\lambda n. i \leq n \wedge n < j)$
by (induct j arbitrary: i) auto

abbreviation $fRestr$:: $('a \times 'a) fset \Rightarrow 'a fset \Rightarrow ('a \times 'a) fset$ **where**
 $fRestr r A \equiv r \upharpoonright (A \times A)$

lift-definition $fId\text{-}on :: 'a \text{ fset} \Rightarrow ('a \times 'a) \text{ fset}$ **is** $Id\text{-}on$
using $Id\text{-}on\text{-}subset\text{-}Times$ $finite\text{-}subset$ **by** $fastforce$

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

lift-definition $fconverse :: ('a \times 'b) \text{ fset} \Rightarrow ('b \times 'a) \text{ fset}$ $(\langle \cdot |^{-1} \rangle)$ $[1000]$ 999
is $converse$ **by** $auto$

lemmas $fconverseI$ $[sym] = converseI$ $[Transfer.transferred]$
lemmas $fconverseD$ $[sym] = converseD$ $[Transfer.transferred]$
lemmas $fconverseE$ $[elim!] = converseE$ $[Transfer.transferred]$
lemmas $fconverse\text{-}iff$ $[iff] = converse\text{-}iff$ $[Transfer.transferred]$
lemmas $fconverse\text{-}fconverse$ $[simp] = converse\text{-}converse$ $[Transfer.transferred]$
lemmas $fconverse\text{-}empty$ $[simp] = converse\text{-}empty$ $[Transfer.transferred]$

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

lemma $finj\text{-}Inl\text{-}Inr$:
 $finj\text{-}on$ Inl A $finj\text{-}on$ Inr A
by $(transfer, auto)+$
lemma $finj\text{-}CInl\text{-}CInr$:
 $finj\text{-}on$ $CInl$ A $finj\text{-}on$ $CInr$ A
using $finj\text{-}Inl\text{-}Inr$ **by** $force+$

lemma $finj\text{-}Some$:
 $finj\text{-}on$ $Some$ A
by $(transfer, auto)$

lift-definition $fImage :: ('a \times 'b) fset \Rightarrow 'a fset \Rightarrow 'b fset$ (**infixr** $\langle |' \rangle 90$) **is**
Image
using *finite-Image* **by force**

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

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

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

lemma *rel-set-cr-fset*:
 $rel\text{-}set\ cr\text{-}fset = (\lambda A B. A = fset \text{ ' } B)$
proof –
have $rel\text{-}set\ cr\text{-}fset\ A\ B \iff A = fset \text{ ' } B$ **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*:
 $pcr\text{-}fset\ cr\text{-}fset = (\lambda x y. x = fset (fset |' y))$
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 $x = \text{sorted-list-of-fset } y \implies x = y$

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) \implies *finite* (*Collect* Q) $\implies \{|x. P\ x \wedge Q\ x|\} = \text{fCollect } P \mid \cap \mid$
fCollect Q

by *auto*

lemma *finite-ntrancl*:

finite $R \implies \text{finite } (\text{ntrancl } n\ R)$

by (*induct* n) *auto*

lift-definition *nftrancl* :: $\text{nat} \Rightarrow ('a \times 'a)\ \text{fset} \Rightarrow ('a \times 'a)\ \text{fset}$ **is** *ntrancl*

by (*intro finite-ntrancl simp*)

lift-definition *frelcomp* :: $('a \times 'b)\ \text{fset} \Rightarrow ('b \times 'c)\ \text{fset} \Rightarrow ('a \times 'c)\ \text{fset}$ (**infixr**

$\langle |O| \rangle$ 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 $| \uparrow R \mid \subseteq \mid S \implies \text{fId-on } S \mid O \mid R = R$

by (*auto intro!*: *frelcompI*)

lemma *fId-on-frelcomp-id2*:

snd $| \uparrow R \mid \subseteq \mid S \implies R \mid O \mid \text{fId-on } S = R$

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* :: $('f, 'v)\ \text{term} \Rightarrow 'f\ \text{fset}$ **is** *funs-term* **using** *finite-funs-term*

by *blast*

lift-definition *fvars-term* :: $('f, 'v)\ \text{term} \Rightarrow 'v\ \text{fset}$ **is** *vars-term* **by** *simp*

lift-definition $fvars-ctxt :: ('f, 'v) ctxt \Rightarrow 'v \text{ fset is vars-ctxt by (simp add: finite-vars-ctxt)}$

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 $ground-fvars-term-empty = ground-vars-term-empty[Transfer.transferred]$

lemma $ffuns-term-Var [simp]: ffuns-term (Var x) = \{\}\}$
by $transfer\ auto$
lemma $ffuns-term-Fun [simp]: ffuns-term (Fun f ts) = |\cup| (ffuns-term |^ fset-of-list ts) |\cup| \{|f|\}$
by $transfer\ auto$

lemma $fvars-term-Var [simp]: fvars-term (Var x) = \{|x|\}$
by $transfer\ auto$
lemma $fvars-term-Fun [simp]: fvars-term (Fun f ts) = |\cup| (fvars-term |^ fset-of-list ts)$
by $transfer\ auto$

lift-definition $ffunas-term :: ('f, 'v) term \Rightarrow ('f \times nat) \text{ fset is funas-term by (simp add: finite-funas-term)}$
lift-definition $ffunas-gterm :: 'f \text{ gterm} \Rightarrow ('f \times nat) \text{ fset is funas-gterm by (simp add: finite-funas-gterm)}$

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 $finite-snd [intro]:$
 $finite\ S \Longrightarrow finite\ \{x. (y, x) \in S\}$
by $(induct\ S\ rule: finite.induct)\ auto$

lemma $finite-Collect-less-eq:$
 $Q \leq P \Longrightarrow finite\ (Collect\ P) \Longrightarrow finite\ (Collect\ Q)$
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 \Rightarrow 'a FSet-Lex-Wrapper \Rightarrow 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' \leq T')

definition *less-FSet-Lex-Wrapper :: 'a FSet-Lex-Wrapper \Rightarrow 'a FSet-Lex-Wrapper \Rightarrow 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

2.6.3 Ground context

datatype (*gfuns-ctxt: 'f*) *gctxt =*
GHole ($\langle \square_G \rangle$) | GMore 'f 'f gterm list 'f gctxt 'f gterm list
declare *gctxt.map-comp[simp]*

fun *gctxt-apply-term :: 'f gctxt \Rightarrow 'f gterm \Rightarrow 'f gterm ($\langle \langle - \rangle_G \rangle$ [1000, 0] 1000)*
where

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

fun *hole-gpos where*

hole-gpos $\square_G = []$ |
hole-gpos (GMore f ss1 C ss2) = length ss1 # hole-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 ⟨oGc⟩ 75) where
  □G oGc D = D |
  (GMore f ss1 C ss2) oGc D = GMore f ss1 (C oGc D) ss2

fun gctxt-at-pos :: 'f gterm ⇒ pos ⇒ 'f gctxt where
  gctxt-at-pos t [] = □G |
  gctxt-at-pos (GFun f ts) (i # ps) =
    GMore f (take i ts) (gctxt-at-pos (ts ! i) ps) (drop (Suc i) ts)

interpretation ctxt-monoid-mult: monoid-mult □G (oGc)
proof
  fix C D E :: 'f gctxt
  show C oGc D oGc E = C oGc (D oGc E) by (induct C) simp-all
  show □G oGc C = C by simp
  show C oGc □G = C by (induct C) simp-all
qed

instantiation gctxt :: (type) monoid-mult
begin
  definition [simp]: 1 = □G
  definition [simp]: (*) = (oGc)
  instance by (intro-classes) (simp-all add: ac-simps)
end

lemma ctxt-ctxt-compose [simp]: (C oGc D)⟨t⟩G = C⟨D⟨t⟩G⟩G
  by (induct C) simp-all

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

fun ctxt-of-gctxt where
  ctxt-of-gctxt □G = □
| ctxt-of-gctxt (GMore f ss C ts) = More f (map term-of-gterm ss) (ctxt-of-gctxt
C) (map term-of-gterm ts)

fun gctxt-of-ctxt where
  gctxt-of-ctxt □ = □G
| gctxt-of-ctxt (More f ss C ts) = GMore f (map gterm-of-term ss) (gctxt-of-ctxt
C) (map gterm-of-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]:

$gctxt\text{-of}\text{-}ctxt (ctxt\text{-of}\text{-}gctxt\ t) = t$

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]:

$ground\text{-}ctxt\ C \implies ctxt\text{-of}\text{-}gctxt (gctxt\text{-of}\text{-}ctxt\ C) = C$

by (induct C) (auto 0 0 intro!: nth-equalityI)

lemma *map-ctxt-of-gctxt* [simp]:

$map\text{-}ctxt\ f\ g (ctxt\text{-of}\text{-}gctxt\ C) = ctxt\text{-of}\text{-}gctxt (map\text{-}gctxt\ f\ C)$

by (induct C) auto

lemma *map-gctxt-of-ctxt* [simp]:

$ground\text{-}ctxt\ C \implies gctxt\text{-of}\text{-}ctxt (map\text{-}ctxt\ f\ g\ C) = map\text{-}gctxt\ f (gctxt\text{-of}\text{-}ctxt\ C)$

by (induct C) auto

lemma *map-gctxt-nempty* [simp]:

$C \neq \square_G \implies map\text{-}gctxt\ f\ C \neq \square_G$

by (cases C) auto

lemma *gctxt-set-funs-ctxt:*

$gfuns\text{-}ctxt\ C = funs\text{-}ctxt (ctxt\text{-of}\text{-}gctxt\ C)$

using *gterm-set-gterm-funs-terms*

by (induct C) fastforce+

lemma *ctxt-set-funs-gctxt:*

assumes *ground-ctxt C*

shows $gfuns\text{-}ctxt (gctxt\text{-of}\text{-}ctxt\ C) = funs\text{-}ctxt\ C$

using *assms term-set-gterm-funs-terms*

by (induct C) fastforce+

lemma *vars-ctxt-of-gctxt* [simp]:

$vars\text{-}ctxt (ctxt\text{-of}\text{-}gctxt\ C) = \{\}$

by (induct C) auto

lemma *vars-ctxt-of-gctxt-subseteq* [simp]:

$vars\text{-}ctxt (ctxt\text{-of}\text{-}gctxt\ C) \subseteq Q \iff True$

by auto

lemma *term-of-gterm-ctxt-apply-ground* [simp]:

$term\text{-of}\text{-}gterm\ s = C\langle l \rangle \implies ground\text{-}ctxt\ C$

$term\text{-of}\text{-}gterm\ s = C\langle l \rangle \implies ground\ l$

by (metis ground-ctxt-apply ground-term-of-gterm)+

lemma *term-of-gterm-ctxt-subst-apply-ground* [simp]:

$term\text{-of}\text{-}gterm\ s = C\langle l \cdot \sigma \rangle \implies x \in vars\text{-}term\ l \implies ground (\sigma\ x)$

by (meson ground-substD term-of-gterm-ctxt-apply-ground(2))

lemma *gctxt-compose-HoleE*:

$C \circ_{Gc} D = \square_G \implies C = \square_G$

$C \circ_{Gc} D = \square_G \implies D = \square_G$

by (cases C; cases D, auto)+

— Relations between ground contexts and contexts

lemma *nempty-ground-ctxt-gctxt* [simp]:

$C \neq \square \implies \text{ground-ctxt } C \implies \text{gctxt-of-ctxt } C \neq \square_G$

by (induct C) auto

lemma *ctxt-of-gctxt-apply* [simp]:

$\text{gterm-of-term } (\text{ctxt-of-gctxt } C) \langle \text{term-of-gterm } t \rangle = C \langle t \rangle_G$

by (induct C) (auto simp: comp-def map-idI)

lemma *ctxt-of-gctxt-apply-gterm*:

$\text{gterm-of-term } (\text{ctxt-of-gctxt } C) \langle t \rangle = C \langle \text{gterm-of-term } t \rangle_G$

by (induct C) (auto simp: comp-def map-idI)

lemma *ground-gctxt-of-ctxt-apply-gterm*:

assumes *ground-ctxt C*

shows $\text{term-of-gterm } (\text{gctxt-of-ctxt } C) \langle t \rangle_G = C \langle \text{term-of-gterm } t \rangle$ 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 $\text{term-of-gterm } (\text{gctxt-of-ctxt } C) \langle \text{gterm-of-term } t \rangle_G = C \langle t \rangle$ using *assms*

by (induct C) (auto simp: comp-def map-idI)

lemma *term-of-gterm-ctxt-apply* [simp]:

$\text{term-of-gterm } s = C \langle l \rangle \implies (\text{gctxt-of-ctxt } C) \langle \text{gterm-of-term } l \rangle_G = s$

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]:

$p \in \text{gpos } t \implies \text{ctxt-of-pos-term } p (\text{term-of-gterm } t) = \text{ctxt-of-gctxt } (\text{gctxt-at-pos } t p)$

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* :: ' $f\ gctxt \Rightarrow pos$ **where**
ghole-pos $\square_G = []$ |
ghole-pos $(GMore\ f\ ss\ D\ ts) = length\ ss\ \#\ ghole\text{-pos}\ D$

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*:
 $ground\text{-ctxt}\ C \Longrightarrow ghole\text{-pos}\ (gctxt\text{-of}\text{-ctxt}\ C) = hole\text{-pos}\ C$
by (*induct* C) *auto*

lemma *gsubt-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 *gsubt-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*]:
 $gsubt\text{-at}\ C\langle u\rangle_G\ (ghole\text{-pos}\ C) = u$
by (*induct* C) *auto*

lemma *gctxt-at-pos-gsubt-at-pos* [*simp*]:
 $p \in gposs\ t \Longrightarrow gsubt\text{-at}\ (gctxt\text{-at}\text{-pos}\ t\ p)\langle u\rangle_G\ p = u$
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-gtxt-at-pos-not-after:

assumes $p \in \text{gposs } t$ $q \in \text{gposs } t \neg (p \leq_p q)$

shows $\text{gfun-at } (\text{gtxt-at-pos } t \ p) \langle v \rangle_G \ q = \text{gfun-at } t \ q$ **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 $j = i \implies \neg (r \leq_p q)$ by auto

from this Cons(2-) Cons(1)[of r gargs t ! j]

have $j = i \implies \text{gfun-at } (\text{gtxt-at-pos } (\text{gargs } t \ ! \ j) \ r) \langle v \rangle_G \ q = \text{gfun-at } (\text{gargs } t \ ! \ j) \ q$

by (cases t) auto

then show ?case **using** Cons(2, 3)

by (cases t) (auto simp: nth-append-Cons min-def)

qed

lemma gposs-ConsE [elim]:

assumes $i \# p \in \text{gposs } t$

obtains f ts where $t = \text{GFun } f \ ts$ $ts \neq []$ $i < \text{length } ts$ $p \in \text{gposs } (ts \ ! \ i)$ **using** *assms*

by (cases t) force+

lemma gposs-gtxt-at-pos-not-after:

assumes $p \in \text{gposs } t$ $q \in \text{gposs } t \neg (p \leq_p q)$

shows $q \in \text{gposs } (\text{gtxt-at-pos } t \ p) \langle v \rangle_G \longleftrightarrow q \in \text{gposs } t$ **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 $j = i \implies \neg (r \leq_p q)$ by auto

from this Cons(2-) Cons(1)[of r gargs t ! j]

have $j = i \implies q \in \text{gposs } (\text{gtxt-at-pos } (\text{gargs } t \ ! \ j) \ r) \langle v \rangle_G \longleftrightarrow q \in \text{gposs } (\text{gargs } t \ ! \ j)$

by (cases t) auto

then show ?case **using** Cons(2, 3)

by (cases t) (auto simp: nth-append-Cons min-def)

qed

lemma gposs-gtxt-at-pos:

$p \in \text{gposs } t \implies \text{gposs } (\text{gtxt-at-pos } t \ p) \langle v \rangle_G = \{q. q \in \text{gposs } t \wedge \neg (p \leq_p q)\} \cup (\text{@}) \ p \ \text{'gposs } v$

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 $\bigwedge q. \neg (p \leq_p q) \implies q \in gposs\ s \longleftrightarrow q \in gposs\ t$
and $(\bigwedge q. q \in gposs\ s \implies \neg (p \leq_p q) \implies gfun\text{-at}\ s\ q = gfun\text{-at}\ t\ q)$
shows $gctxt\text{-at-pos}\ s\ p = gctxt\text{-at-pos}\ t\ p$ **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* :: $'f\ gctxt \Rightarrow ('f \times nat)\ set$ **where**
funas-gctxt GHole = {} |
funas-gctxt (GMore f ss1 D ss2) = {(f, Suc (length (ss1 @ ss2)))}
\cup funas-gctxt D \cup \bigcup (set (map funas-gterm (ss1 @ ss2)))

lemma *funas-gctxt-of-ctxt [simp]:*
 $ground\text{-ctxt}\ C \implies funas\text{-gctxt}\ (gctxt\text{-of-ctxt}\ C) = funas\text{-ctxt}\ C$
by (*induct C*) (*auto simp: funas-gterm-gterm-of-term*)

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

lemma *inj-gctxt-of-ctxt-on-ground:*
 $inj\text{-on}\ gctxt\text{-of-ctxt}\ (Collect\ ground\text{-ctxt})$
using *gctxt-of-ctxt-inv* **by** (*fastforce simp: inj-on-def*)

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

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

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

2.6.4 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*]: $(s, t) \in \mathcal{R} \implies (s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$
| *step* [*intro*]: $\text{length } ss = \text{length } ts \implies (\forall i < \text{length } ts. (ss ! i, ts ! i) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}) \implies (f, \text{length } ss) \in \mathcal{F} \implies$
 $(\text{GFun } f \ ss, \text{GFun } f \ ts) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$

lemma *gmctxt-cl-idemp* [*simp*]:
 $\text{gmctxt-cl } \mathcal{F} (\text{gmctxt-cl } \mathcal{F} \mathcal{R}) = \text{gmctxt-cl } \mathcal{F} \mathcal{R}$

proof –

{**fix** *s t* **assume** $(s, t) \in \text{gmctxt-cl } \mathcal{F} (\text{gmctxt-cl } \mathcal{F} \mathcal{R})$
then have $(s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$
by (*induct*) (*auto intro: gmctxt-cl.step*)}
then show *?thesis* **by** *auto*
qed

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

lemma *gmctxt-cl-swap*:
 $\text{gmctxt-cl } \mathcal{F} (\text{prod.swap } ' \mathcal{R}) = \text{prod.swap } ' \text{gmctxt-cl } \mathcal{F} \mathcal{R}$ (**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 $(t, s) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$ **by** *auto*
then have $(s, t) \in ?Ls$ **by** *induct auto*}
ultimately show *?thesis* **by** *auto*
qed

lemma *gmctxt-cl-mono-fun*:
assumes $\mathcal{F} \subseteq \mathcal{G}$ **shows** $\text{gmctxt-cl } \mathcal{F} \mathcal{R} \subseteq \text{gmctxt-cl } \mathcal{G} \mathcal{R}$

proof –

{**fix** *s t* **assume** $(s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$ **then have** $(s, t) \in \text{gmctxt-cl } \mathcal{G} \mathcal{R}$
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** $\text{gmctxt-cl } \mathcal{F} \mathcal{P} \subseteq \text{gmctxt-cl } \mathcal{F} \mathcal{R}$

proof –

{**fix** *s t* **assume** $(s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{P}$ **then have** $(s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$ **using**
assms
by *induct auto*}

then show *?thesis* **by** *auto*
qed

definition *gcomp-rel* :: ('f × nat) set ⇒ 'f gterm rel ⇒ 'f gterm rel ⇒ 'f gterm rel **where**

$$gcomp-rel \mathcal{F} \mathcal{R} \mathcal{S} = (\mathcal{R} \ O \ gmctxt-cl \ \mathcal{F} \ \mathcal{S}) \cup (gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ O \ \mathcal{S})$$

definition *grancl-rel* :: ('f × nat) set ⇒ 'f gterm rel ⇒ 'f gterm rel **where**

$$grancl-rel \ \mathcal{F} \ \mathcal{R} = (gmctxt-cl \ \mathcal{F} \ \mathcal{R})^+ \ O \ \mathcal{R} \ O \ (gmctxt-cl \ \mathcal{F} \ \mathcal{R})^+$$

lemma *gcomp-rel*:

$$gmctxt-cl \ \mathcal{F} \ (gcomp-rel \ \mathcal{F} \ \mathcal{R} \ \mathcal{S}) = gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ O \ gmctxt-cl \ \mathcal{F} \ \mathcal{S} \ (\text{is } ?Ls = ?Rs)$$

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* - $\lambda u i. (ss ! i, u) \in gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ \wedge \ (u, ts ! i) \in gmctxt-cl \ \mathcal{F} \ \mathcal{S}$]

obtain *us* **where** *l*: *length us* = *length ts* **and**

inv: $\forall i < \text{length } ts. (ss ! i, us ! i) \in gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ \wedge \ (us ! i, ts ! i) \in gmctxt-cl \ \mathcal{F} \ \mathcal{S}$

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*) $\in \mathcal{S}$)

case *True*

then have (*GFun f ss, u*) $\in gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ O \ \mathcal{S}$ **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 $i < \text{length } us \implies$

$(ss ! i, us ! i) \in gmctxt-cl \ \mathcal{F} \ (\mathcal{R} \ O \ gmctxt-cl \ \mathcal{F} \ \mathcal{S} \cup gmctxt-cl \ \mathcal{F} \ \mathcal{R} \ O \ \mathcal{S})$

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  $\subseteq$  ?Ls
  by (auto simp: gcomp-rel-def)
qed

```

2.6.5 Signature closed property

definition *all-ctxt-closed-gterm* :: $(f \times \text{nat}) \text{ set} \Rightarrow f \text{ gterm rel} \Rightarrow \text{bool}$ **where**
all-ctxt-closed-gterm $F r \iff (\forall f \text{ ts ss. } (f, \text{length ss}) \in F \longrightarrow \text{length ss} = \text{length ts} \longrightarrow$
 $(\forall i. i < \text{length ts} \longrightarrow (ss ! i, ts ! i) \in r) \longrightarrow$
 $(\text{GFun } f \text{ ss, GFun } f \text{ ts}) \in r)$

lemma *all-ctxt-closed-gtermI*:
assumes $\bigwedge f \text{ ss ts. } (f, \text{length ss}) \in \mathcal{F} \implies \text{length ss} = \text{length ts} \implies$
 $(\forall i < \text{length ts. } (ss ! i, ts ! i) \in r) \implies (\text{GFun } f \text{ ss, GFun } f \text{ ts}) \in r$
shows *all-ctxt-closed-gterm* $\mathcal{F} r$ **using** *assms*
unfolding *all-ctxt-closed-gterm-def* **by** *auto*

lemma *all-ctxt-closed-gtermD*:
all-ctxt-closed-gterm $F r \implies (f, \text{length ss}) \in F \implies \text{length ss} = \text{length ts} \implies$
 $(\forall i < \text{length ts. } (ss ! i, ts ! i) \in r) \implies (\text{GFun } f \text{ ss, GFun } f \text{ ts}) \in r$
by (*auto simp: all-ctxt-closed-gterm-def*)

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

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

lemma *all-ctxt-closed-gterm-gmctxt-cl-idem* [*simp*]:
assumes *all-ctxt-closed-gterm* $\mathcal{F} \mathcal{R}$
shows $\text{gmctxt-cl } \mathcal{F} \mathcal{R} = \mathcal{R}$

proof –
{fix $s \ t$ **assume** $(s, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$ **then have** $(s, t) \in \mathcal{R}$
proof (*induct*)
case (*step ss ts f*)
show ?*case* **using** *step*(2) *all-ctxt-closed-gtermD*[*OF assms step*(3, 1)]
by *auto*
qed *auto*}
then show ?*thesis* **by** *auto*

qed

2.6.6 Transitive closure preserves *all-ctxt-closed-gterm*

induction scheme for transitive closures of lists

inductive-set *trancl-list* for \mathcal{R} where

base[*intro*, *Pure.intro*] : $\text{length } xs = \text{length } ys \implies$
 $(\forall i < \text{length } ys. (xs ! i, ys ! i) \in \mathcal{R}) \implies (xs, ys) \in \text{trancl-list } \mathcal{R}$
| *list-trancl* [*Pure.intro*]: $(xs, ys) \in \text{trancl-list } \mathcal{R} \implies i < \text{length } ys \implies (ys ! i, z)$
 $\in \mathcal{R} \implies$
 $(xs, ys[i := z]) \in \text{trancl-list } \mathcal{R}$

lemma *trancl-list-appendI* [*simp*, *intro*]:

$(xs, ys) \in \text{trancl-list } \mathcal{R} \implies (x, y) \in \mathcal{R} \implies (x \# xs, y \# ys) \in \text{trancl-list } \mathcal{R}$

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** *: $y \# ys[i := z] = (y \# ys)[\text{Suc } i := z]$ **by** *auto*

show *?case* **using** *list-trancl unfolding **

by (*intro trancl-list.list-trancl*) *auto*

qed

lemma *trancl-list-append-tranclI* [*intro*]:

$(x, y) \in \mathcal{R}^+ \implies (xs, ys) \in \text{trancl-list } \mathcal{R} \implies (x \# xs, y \# ys) \in \text{trancl-list } \mathcal{R}$

proof (*induct rule: trancl.induct*)

case (*trancl-into-trancl a b c*)

then have $(a \# xs, b \# ys) \in \text{trancl-list } \mathcal{R}$ **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*:

$(xs, ys) \in \text{trancl-list } \mathcal{R} \iff \text{length } xs = \text{length } ys \wedge (\forall i < \text{length } ys. (xs ! i, ys ! i) \in \mathcal{R}^+)$ (**is** $?Ls \iff ?Rs$)

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) ∈  $\mathcal{R}^+$ 
  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 $\text{length } ss = \text{length } ts \ \forall \ i < \text{length } ts. (ss ! i, ts ! i) \in \mathcal{R}^+$
 and $\bigwedge xs \ ys. \text{length } xs = \text{length } ys \implies \forall \ i < \text{length } ys. (xs ! i, ys ! i) \in \mathcal{R} \implies P \ xs \ ys$
 and $\bigwedge xs \ ys \ i \ z. \text{length } xs = \text{length } ys \implies \forall \ i < \text{length } ys. (xs ! i, ys ! i) \in \mathcal{R}^+ \implies P \ xs \ ys$
 $\implies i < \text{length } ys \implies (ys ! i, z) \in \mathcal{R} \implies P \ xs \ (ys[i := z])$
 shows $P \ ss \ ts$ using *assms*
 by (intro trancl-list.induct[of ss ts $\mathcal{R} \ P$]) (auto simp: trancl-list-conv)

lemma *trancl-list-all-step-induct* [consumes 2, case-names base step]:
 assumes $\text{length } ss = \text{length } ts \ \forall \ i < \text{length } ts. (ss ! i, ts ! i) \in \mathcal{R}^+$
 and base: $\bigwedge xs \ ys. \text{length } xs = \text{length } ys \implies \forall \ i < \text{length } ys. (xs ! i, ys ! i) \in \mathcal{R} \implies P \ xs \ ys$
 and steps: $\bigwedge xs \ ys \ zs. \text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies \forall \ i < \text{length } zs. (xs ! i, ys ! i) \in \mathcal{R}^+ \implies \forall \ i < \text{length } zs. (ys ! i, zs ! i) \in \mathcal{R} \vee ys ! i = zs ! i \implies P \ xs \ ys \implies P \ xs \ zs$
 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-gterm-trancl*:
 assumes $\text{all-ctxt-closed-gterm } \mathcal{F} \ \mathcal{R} \ \mathcal{R} \subseteq \mathcal{T}_G \ \mathcal{F} \times \mathcal{T}_G \ \mathcal{F}$
 shows $\text{all-ctxt-closed-gterm } \mathcal{F} \ (\mathcal{R}^+)$
proof –
 {fix $f \ ss \ ts$ assume $\text{sig}: (f, \text{length } ss) \in \mathcal{F}$ and
 steps: $\text{length } ss = \text{length } ts \ \forall \ i < \text{length } ts. (ss ! i, ts ! i) \in \mathcal{R}^+$
 have $(GFun \ f \ ss, GFun \ f \ ts) \in \mathcal{R}^+$ using steps sig
proof (induct rule: trancl-list-induct)
 case (base ss ts)
 then show ?case using *all-ctxt-closed-gtermD*[OF *assms*(1) base(3, 1, 2)]
 by auto
 next

```

    case (step ss ts i t')
  from step(2) have j < length ts  $\implies$  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-gterm-refl-on[OF assms(1)]]
  have (GFun f ts, GFun f (ts[i := t']))  $\in$   $\mathcal{R}$  using step(1, 4-)
    by (intro all-ctxt-closed-gtermD[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-closed-gtermI)
qed

end
theory Horn-Inference
  imports Main
begin

datatype 'a horn = horn 'a list 'a (infix <math>\rightarrow_h</math> 55)

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

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

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

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

inductive step :: 'a set  $\times$  'a set  $\Rightarrow$  'a set  $\times$  'a set  $\Rightarrow$  bool (infix <math>\vdash</math> 50) where
  delete:  $x \in B \implies (\text{insert } x G, B) \vdash (G, B)$ 
| propagate:  $(\text{insert } x G, B) \vdash (G \cup \text{infer1 } x B, \text{insert } x B)$ 
| refl:  $(G, B) \vdash (G, B)$ 
| trans:  $(G, B) \vdash (G', B') \implies (G', B') \vdash (G'', B'') \implies (G, B) \vdash (G'', B'')$ 

lemma step-mono:
   $(G, B) \vdash (G', B') \implies (H \cup G, B) \vdash (H \cup G', B')$ 
  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)  $\iff G \subseteq \text{saturate} \wedge B \subseteq \text{saturate} \wedge (\forall a \text{ as. as } \rightarrow_h a \in \mathcal{H} \wedge \text{set } as \subseteq B \longrightarrow a \in G \cup B)$ 

lemma inv-start:
  shows invariant (infer0, {})

```

```

    by (auto simp: infer0-def invariant.simps intro: infer)

lemma inv-step:
  assumes invariant (G, B) (G, B) ⊢ (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 ∪ local.infer1 x B and ?B' = insert x B
  have ?G' ⊆ saturate ?B' ⊆ saturate
    using assms(1) propagate by (auto 0 3 simp: infer1-def intro: saturate.infer)
  moreover have as →h a ∈ ℋ ⇒ set as ⊆ ?B' ⇒ a ∈ ?G' ∪ ?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, {}) ⊢ ({}, B) ⇒ B = saturate
  by (metis inv-start inv-step inv-end)

end

lemma horn-infer0-union:
  horn.infer0 (ℋ1 ∪ ℋ2) = horn.infer0 ℋ1 ∪ horn.infer0 ℋ2
  by (auto simp: horn.infer0-def)

lemma horn-infer1-union:
  horn.infer1 (ℋ1 ∪ ℋ2) x B = horn.infer1 ℋ1 x B ∪ horn.infer1 ℋ2 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 (λx a. 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 list ⇒ ('a list) option where
  saturate-rec x bs = (if x ∈ set bs then Some bs else
    fold (λx. case-option None (saturate-rec x)) (infer1-impl x bs) (Some (x #
  bs)))

definition saturate-impl where
  saturate-impl = fold (λx. 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*) \vdash ($\{\}$, *set bs'*)

if *fold* ($\lambda x a. \text{case } a \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } a \Rightarrow \text{saturate-rec } x a$) *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 \neq None*

proof –

have $*$: *fold* ($\lambda x. \text{case-option } \text{None} (\text{saturate-rec } x)$) *ds* (*Some bs*) $\neq \text{None}$

if *set bs \subseteq saturate set ds \subseteq 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) \subseteq 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 \notin 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-Utills
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: ∧x bs. infer1 x (fset bs) = set (infer1-impl x bs)
begin

lemma saturate-rec-sound:
  saturate-rec x bs = Some bs' ⇒ ({x}, fset bs) ⊢ ({}, fset 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 ( $\exists$  rec)
  have [dest!]: (set xs, fset ys)  $\vdash$  ({}, fset bs $\wedge$ )
    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  $\exists$ (1)], of a ys - set xs {} fset bs $\wedge$ ]
Cons
    by (cases rec a ys) auto
  qed (auto intro: refl)
  show ?case using propagate[of x {} fset bs, unfolded infer1 Un-empty-left]  $\exists$ (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 $\wedge$ ]
Cons
    by (cases saturate-rec a ys) auto
  qed (auto intro: refl)
  from this[of infer0-impl {||} B $\wedge$ ] assms step-sound show ?thesis
    by (auto simp: saturate-impl-def infer0)
qed

```

```

lemma saturate-impl-complete:
  assumes finite saturate
  shows saturate-impl  $\neq$  None
proof -
  have *: fold ( $\lambda x.$  case-option None (saturate-rec x)) ds (Some bs)  $\neq$  None
    if fset bs  $\subseteq$  saturate set ds  $\subseteq$  saturate for bs ds
    using that
  proof (induct card (saturate - fset bs) arbitrary: bs ds rule: less-induct)
    case less
    show ?case using less( $\exists$ )
  proof (induct ds)
    case (Cons d ds)
    have infer1 d (fset bs)  $\subseteq$  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-Utills
  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 × '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 Δ ≡ |U| ((fset-of-list ∘ r-lhs-states)
|↑ Δ)

```

abbreviation *rule-target-states* **where** $rule\text{-}target\text{-}states \Delta \equiv (r\text{-}rhs \mid^{\dagger} \Delta)$
definition *rule-states* **where** $rule\text{-}states \Delta \equiv rule\text{-}arg\text{-}states \Delta \mid \cup \mid rule\text{-}target\text{-}states \Delta$

definition *eps-states* **where** $eps\text{-}states \Delta_{\varepsilon} \equiv (fst \mid^{\dagger} \Delta_{\varepsilon}) \mid \cup \mid (snd \mid^{\dagger} \Delta_{\varepsilon})$
definition $\mathcal{Q} \mathcal{A} = rule\text{-}states (rules \mathcal{A}) \mid \cup \mid eps\text{-}states (eps \mathcal{A})$
abbreviation $\mathcal{Q}_r \mathcal{A} \equiv \mathcal{Q} (ta \mathcal{A})$

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

definition $ta\text{-}sig \mathcal{A} = (\lambda r. (r\text{-}root \ r, length (r\text{-}lhs\text{-}states \ r))) \mid^{\dagger} (rules \ \mathcal{A})$

3.1.1 Rechability of a term induced by a tree automaton

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

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

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* $\equiv fst \circ groot\text{-}sym$
abbreviation *ex-comp-state* $\equiv snd \circ groot\text{-}sym$

inductive *run* **for** \mathcal{A} **where**
 $step: length \ qs = length \ ts \implies (\forall i < length \ ts. run \ \mathcal{A} (qs \ ! \ i) (ts \ ! \ i)) \implies TA\text{-}rule \ f (map \ ex\text{-}comp\text{-}state \ qs) \ q \mid \in \mid (rules \ \mathcal{A}) \implies (q = q' \vee (q, q') \mid \in \mid (eps \ \mathcal{A})^{+} \mid) \implies run \ \mathcal{A} (GFun (q, q') \ qs) (GFun \ f \ ts)$

3.1.2 Language acceptance

definition *ta-lang* $:: 'q \text{ fset} \Rightarrow ('q, 'f) ta \Rightarrow ('f, 'v) terms$ **where**
 $[code \ del]: ta\text{-}lang \ \mathcal{Q} \ \mathcal{A} = \{ adapt\text{-}vars \ t \mid t. ground \ t \wedge \mathcal{Q} \mid \cap \mid ta\text{-}der \ \mathcal{A} \ t \neq \{ \mid \} \}$

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

definition *gta-lang* **where**

$$gta\text{-}lang\ \mathcal{Q}\ \mathcal{A} = \{t. \mathcal{Q} \mid \cap \mid gta\text{-}der\ \mathcal{A}\ t \neq \{\mid\}\}$$

definition \mathcal{L} **where**

$$\mathcal{L}\ \mathcal{A} = gta\text{-}lang\ (fin\ \mathcal{A})\ (ta\ \mathcal{A})$$

definition *reg-Restr- \mathcal{Q}_f* **where**

$$reg\text{-}Restr\text{-}\mathcal{Q}_f\ R = Reg\ (fin\ R \mid \cap \mid \mathcal{Q}_r\ R)\ (ta\ R)$$

3.1.3 Trimming

definition *ta-restrict* **where**

$$ta\text{-}restrict\ \mathcal{A}\ \mathcal{Q} = TA\ \{\mid TA\text{-}rule\ f\ qs\ q \mid f\ qs\ q. TA\text{-}rule\ f\ qs\ q \mid \in \mid rules\ \mathcal{A} \wedge fset\text{-}of\text{-}list\ qs \mid \subseteq \mid \mathcal{Q} \wedge q \mid \in \mid \mathcal{Q} \mid \} (fRestr\ (eps\ \mathcal{A})\ \mathcal{Q})$$

definition *ta-reachable* $:: ('q, 'f)\ ta \Rightarrow 'q\ fset$ **where**

$$ta\text{-}reachable\ \mathcal{A} = \{\mid q \mid q. \exists t. ground\ t \wedge q \mid \in \mid ta\text{-}der\ \mathcal{A}\ t \mid \}$$

definition *ta-productive* $:: 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow 'q\ fset$ **where**

$$ta\text{-}productive\ P\ \mathcal{A} \equiv \{\mid q \mid q\ q' C. q' \mid \in \mid ta\text{-}der\ \mathcal{A}\ (C\langle Var\ q \rangle) \wedge q' \mid \in \mid P \mid \}$$

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

definition *ta-is-trim* $:: 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow bool$ **where**

$$ta\text{-}is\text{-}trim\ P\ \mathcal{A} \equiv \forall q. q \mid \in \mid \mathcal{Q}\ \mathcal{A} \longrightarrow q \mid \in \mid ta\text{-}reachable\ \mathcal{A} \wedge q \mid \in \mid ta\text{-}productive\ P\ \mathcal{A}$$

definition *reg-is-trim* $:: ('q, 'f)\ reg \Rightarrow bool$ **where**

$$reg\text{-}is\text{-}trim\ R \equiv ta\text{-}is\text{-}trim\ (fin\ R)\ (ta\ R)$$

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

abbreviation *ta-only-reach* $:: ('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta$ **where**

$$ta\text{-}only\text{-}reach\ \mathcal{A} \equiv ta\text{-}restrict\ \mathcal{A}\ (ta\text{-}reachable\ \mathcal{A})$$

abbreviation *ta-only-prod* $:: 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta$ **where**

$$ta\text{-}only\text{-}prod\ P\ \mathcal{A} \equiv ta\text{-}restrict\ \mathcal{A}\ (ta\text{-}productive\ P\ \mathcal{A})$$

definition *reg-reach* **where**

$$reg\text{-}reach\ R = Reg\ (fin\ R)\ (ta\text{-}only\text{-}reach\ (ta\ R))$$

definition *reg-prod* **where**

$$reg\text{-}prod\ R = Reg\ (fin\ R)\ (ta\text{-}only\text{-}prod\ (fin\ R)\ (ta\ R))$$

definition *trim-ta* $:: 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta$ **where**

$$trim\text{-}ta\ P\ \mathcal{A} = ta\text{-}only\text{-}prod\ P\ (ta\text{-}only\text{-}reach\ \mathcal{A})$$

definition *trim-reg* **where**

$$trim\text{-}reg\ R = Reg\ (fin\ R)\ (trim\text{-}ta\ (fin\ R)\ (ta\ R))$$

3.1.4 Mapping over tree automata

definition $fmap\text{-states}\text{-}ta :: ('a \Rightarrow 'b) \Rightarrow ('a, 'f) ta \Rightarrow ('b, 'f) ta$ **where**
 $fmap\text{-states}\text{-}ta f \mathcal{A} = TA (map\text{-}ta\text{-}rule f id \mid \mid \mid \text{rules } \mathcal{A}) (map\text{-}both f \mid \mid \mid \text{eps } \mathcal{A})$

definition $fmap\text{-funs}\text{-}ta :: ('f \Rightarrow 'g) \Rightarrow ('a, 'f) ta \Rightarrow ('a, 'g) ta$ **where**
 $fmap\text{-funs}\text{-}ta f \mathcal{A} = TA (map\text{-}ta\text{-}rule id f \mid \mid \mid \text{rules } \mathcal{A}) (\text{eps } \mathcal{A})$

definition $fmap\text{-states}\text{-}reg :: ('a \Rightarrow 'b) \Rightarrow ('a, 'f) reg \Rightarrow ('b, 'f) reg$ **where**
 $fmap\text{-states}\text{-}reg f R = Reg (f \mid \mid \mid \text{fin } R) (fmap\text{-states}\text{-}ta f (ta R))$

definition $fmap\text{-funs}\text{-}reg :: ('f \Rightarrow 'g) \Rightarrow ('a, 'f) reg \Rightarrow ('a, 'g) reg$ **where**
 $fmap\text{-funs}\text{-}reg f R = Reg (\text{fin } R) (fmap\text{-funs}\text{-}ta f (ta R))$

3.1.5 Product construction (language intersection)

definition $prod\text{-}ta\text{-}rules :: ('q1, 'f) ta \Rightarrow ('q2, 'f) ta \Rightarrow ('q1 \times 'q2, 'f) ta\text{-}rule$ *fset*
where

$prod\text{-}ta\text{-}rules \mathcal{A} \mathcal{B} = \{ \mid TA\text{-}rule f qs q \mid f qs q. TA\text{-}rule f (map fst qs) (fst q) \mid \in \mid \text{rules } \mathcal{A} \wedge$

$TA\text{-}rule f (map snd qs) (snd q) \mid \in \mid \text{rules } \mathcal{B} \}$

declare $prod\text{-}ta\text{-}rules\text{-}def$ [*simp*]

definition $prod\text{-}epsLp$ **where**

$prod\text{-}epsLp \mathcal{A} \mathcal{B} = (\lambda (p, q). (fst p, fst q) \mid \in \mid \text{eps } \mathcal{A} \wedge snd p = snd q \wedge snd q \mid \in \mid \mathcal{Q} \mathcal{B})$

definition $prod\text{-}epsRp$ **where**

$prod\text{-}epsRp \mathcal{A} \mathcal{B} = (\lambda (p, q). (snd p, snd q) \mid \in \mid \text{eps } \mathcal{B} \wedge fst p = fst q \wedge fst q \mid \in \mid \mathcal{Q} \mathcal{A})$

definition $prod\text{-}ta :: ('q1, 'f) ta \Rightarrow ('q2, 'f) ta \Rightarrow ('q1 \times 'q2, 'f) ta$ **where**

$prod\text{-}ta \mathcal{A} \mathcal{B} = TA (prod\text{-}ta\text{-}rules \mathcal{A} \mathcal{B})$

$(fCollect (prod\text{-}epsLp \mathcal{A} \mathcal{B}) \mid \cup \mid fCollect (prod\text{-}epsRp \mathcal{A} \mathcal{B}))$

definition $reg\text{-}intersect$ **where**

$reg\text{-}intersect R L = Reg (\text{fin } R \mid \times \mid \text{fin } L) (prod\text{-}ta (ta R) (ta L))$

3.1.6 Union construction (language union)

definition $ta\text{-}union$ **where**

$ta\text{-}union \mathcal{A} \mathcal{B} = TA (\text{rules } \mathcal{A} \mid \cup \mid \text{rules } \mathcal{B}) (\text{eps } \mathcal{A} \mid \cup \mid \text{eps } \mathcal{B})$

definition $reg\text{-}union$ **where**

$reg\text{-}union R L = Reg (\text{Inl } \mid \mid \mid (\text{fin } R \mid \cap \mid \mathcal{Q}_r R) \mid \cup \mid \text{Inr } \mid \mid \mid (\text{fin } L \mid \cap \mid \mathcal{Q}_r L))$

$(ta\text{-}union (fmap\text{-states}\text{-}ta \text{Inl } (ta R)) (fmap\text{-states}\text{-}ta \text{Inr } (ta L)))$

3.1.7 Epsilon free and tree automaton accepting empty language

definition $eps\text{-}free\text{-}rulep$ **where**

$eps\text{-free-rulep } \mathcal{A} = (\lambda r. \exists f qs q'. r = TA\text{-rule } f qs q' \wedge TA\text{-rule } f qs q \mid \in \mid \text{ rules } \mathcal{A} \wedge (q = q' \vee (q, q') \mid \in \mid (eps \mathcal{A})^+))$

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

definition $is\text{-ta-eps-free} :: ('q, 'f) ta \Rightarrow bool$ **where**
 $is\text{-ta-eps-free } \mathcal{A} \longleftrightarrow eps \mathcal{A} = \{\mid\mid\}$

definition $ta\text{-empty} :: 'q fset \Rightarrow ('q, 'f) ta \Rightarrow bool$ **where**
 $ta\text{-empty } Q \mathcal{A} \longleftrightarrow ta\text{-reachable } \mathcal{A} \mid \cap \mid Q \mid \subseteq \mid \{\mid\mid\}$

definition $eps\text{-free-reg}$ **where**
 $eps\text{-free-reg } R = Reg (fin R) (eps\text{-free } (ta R))$

definition $reg\text{-empty}$ **where**
 $reg\text{-empty } R = ta\text{-empty } (fin R) (ta R)$

3.1.8 Relabeling tree automaton states to natural numbers

definition $map\text{-fset-to-nat} :: ('a :: linorder) fset \Rightarrow 'a \Rightarrow nat$ **where**
 $map\text{-fset-to-nat } X = (\lambda x. the (mem\text{-idx } x (sorted\text{-list-of-fset } X)))$

definition $map\text{-fset-fset-to-nat} :: ('a :: linorder) fset fset \Rightarrow 'a fset \Rightarrow nat$ **where**
 $map\text{-fset-fset-to-nat } X = (\lambda x. the (mem\text{-idx } (sorted\text{-list-of-fset } x) (sorted\text{-list-of-fset } \mid \uparrow \mid X))))$

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

definition $relabel\text{-}Q_f :: ('q :: linorder) fset \Rightarrow ('q :: linorder, 'f) ta \Rightarrow nat fset$
where
 $relabel\text{-}Q_f Q \mathcal{A} = map\text{-fset-to-nat } (Q \mathcal{A}) \mid \uparrow \mid (Q \mid \cap \mid Q \mathcal{A})$

definition $relabel\text{-reg} :: ('q :: linorder, 'f) reg \Rightarrow (nat, 'f) reg$ **where**
 $relabel\text{-reg } R = Reg (relabel\text{-}Q_f (fin R) (ta R)) (relabel\text{-ta } (ta R))$

— The instantiation of $<$ and \leq for finite sets are $\mid \subset \mid$ and $\mid \subseteq \mid$ which don't give rise to a total order and therefore it cannot be an instance of the type class `linorder`. However taking the lexicographic 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 $relabel\text{-fset-ta} :: (('q :: linorder) fset, 'f) ta \Rightarrow (nat, 'f) ta$ **where**
 $relabel\text{-fset-ta } \mathcal{A} = fmap\text{-states-ta } (map\text{-fset-fset-to-nat } (Q \mathcal{A})) \mathcal{A}$

definition $relabel\text{-fset-}Q_f :: ('q :: linorder) fset fset \Rightarrow (('q :: linorder) fset, 'f) ta \Rightarrow nat fset$ **where**
 $relabel\text{-fset-}Q_f Q \mathcal{A} = map\text{-fset-fset-to-nat } (Q \mathcal{A}) \mid \uparrow \mid (Q \mid \cap \mid Q \mathcal{A})$

definition *relabel-fset-reg* :: (('q :: linorder) fset, 'f) reg \Rightarrow (nat, 'f) reg **where**
relabel-fset-reg R = Reg (relabel-fset-Q_f (fin R) (ta R)) (relabel-fset-ta (ta R))

definition *srules* A = fset (rules A)

definition *seps* A = fset (eps 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 A = *rules* B \Longrightarrow *eps* A = *eps* B \Longrightarrow A = B
using *ta.expand* **by** *blast*

lemma *rule-states-code* [code]:

rule-states Δ = |∪| ((λ r. *fininsert* (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_ε = |∪| ((λ (q, q'). {|q, q'|}) |∣ Δ_ε) (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 |∪| Γ) = *rule-states* Δ |∪| *rule-states* Γ
unfolding *rule-states-def*
by *fastforce*

lemma *rule-states-mono*:

Δ |⊆| Γ \Longrightarrow *rule-states* Δ |⊆| *rule-states* Γ
unfolding *rule-states-def*
by *force*

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

eps-states (Δ |∪| Γ) = *eps-states* Δ |∪| *eps-states* Γ

unfolding *eps-states-def*
by *auto*

lemma *eps-states-image* [*simp*]:
 $eps\text{-states} (map\text{-both } f \mid^! \Delta_\varepsilon) = f \mid^! eps\text{-states } \Delta_\varepsilon$
unfolding *eps-states-def map-prod-def*
by (*force simp: fimage-iff*)

lemma *eps-states-mono*:
 $\Delta \mid\subseteq\mid \Gamma \implies eps\text{-states } \Delta \mid\subseteq\mid eps\text{-states } \Gamma$
unfolding *eps-states-def*
by *transfer auto*

lemma *eps-statesI* [*intro*]:
 $(p, q) \mid\in\mid \Delta \implies p \mid\in\mid eps\text{-states } \Delta$
 $(p, q) \mid\in\mid \Delta \implies q \mid\in\mid eps\text{-states } \Delta$
unfolding *eps-states-def*
by (*auto simp add: rev-image-eqI*)

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

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

lemma *rule-statesI* [*intro*]:
assumes $r \mid\in\mid \Delta \wedge q \mid\in\mid finsert (r\text{-rhs } r) (fset\text{-of-list } (r\text{-lhs-states } r))$
shows $q \mid\in\mid rule\text{-states } \Delta$ **using** *assms*
by (*auto simp: rule-states-def*)

Destruction rule for states

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

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

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

lemma *eps-trancl-statesD*:
 $(p, q) \mid \in \mid (\text{eps } \mathcal{A}) \mid^+ \implies p \mid \in \mid \mathcal{Q} \mathcal{A} \wedge q \mid \in \mid \mathcal{Q} \mathcal{A}$
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*:
 $\text{finite } \{ \text{TA-rule } f \text{ } q \text{ } q \mid f \text{ } q \text{ } q. \text{ TA-rule } f \text{ } q \text{ } q \mid \in \mid \text{rules } \mathcal{A} \}$ (**is finite** ?*S*)
proof –
have $\{ f \text{ } q \text{ } \rightarrow q \mid f \text{ } q \text{ } q. f \text{ } q \text{ } \rightarrow q \mid \in \mid \text{rules } \mathcal{A} \} \subseteq \text{fset } (\text{rules } \mathcal{A})$
by *auto*
from *finite-subset[OF this]* **show** ?*thesis* **by** *simp*
qed

lemma *map-ta-rule-finite*:
 $\text{finite } \Delta \implies \text{finite } \{ \text{TA-rule } (g \text{ } h) (\text{map } f \text{ } q \text{ } s) (f \text{ } q) \mid h \text{ } q \text{ } s. \text{ TA-rule } h \text{ } q \text{ } s \text{ } q \in \Delta \}$
proof (*induct rule: finite.induct*)
case (*insertI A a*)
have *union*: $\{ \text{TA-rule } (g \text{ } h) (\text{map } f \text{ } q \text{ } s) (f \text{ } q) \mid h \text{ } q \text{ } s. \text{ TA-rule } h \text{ } q \text{ } s \text{ } q \in \text{insert } a \text{ } A \} =$
 $\{ \text{TA-rule } (g \text{ } h) (\text{map } f \text{ } q \text{ } s) (f \text{ } q) \mid h \text{ } q \text{ } s. \text{ TA-rule } h \text{ } q \text{ } s \text{ } q = a \} \cup \{ \text{TA-rule } (g \text{ } h) (\text{map } f \text{ } q \text{ } s) (f \text{ } q) \mid h \text{ } q \text{ } s. \text{ TA-rule } h \text{ } q \text{ } s \text{ } q \in A \}$
by *auto*
have *finite* $\{ g \text{ } h \text{ } \text{map } f \text{ } q \text{ } s \rightarrow f \text{ } q \mid h \text{ } q \text{ } s. h \text{ } q \text{ } s \rightarrow q = a \}$
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*:
 $\text{map-ta-rule } f \text{ } g \circ \text{map-ta-rule } f' \text{ } g' = \text{map-ta-rule } (f \circ f') (g \circ g')$
using *ta-rule.map-comp*[*of f g*]
by (*auto simp: comp-def*)

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

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

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

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

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

lemma *finite-states*:
 $finite\ \{q. \exists f\ p\ ps. f\ ps \rightarrow p\ |\in|\ rules\ \mathcal{A} \wedge (p = q \vee (p, q)\ |\in|\ (eps\ \mathcal{A})^+)\}$ (**is**
 $finite\ ?set$)
proof –
have $?set \subseteq fset\ (Q\ \mathcal{A})$
by (*intro subsetI, drule CollectD*)
 $(metis\ eps\ trancl\ statesD\ rule\text{-}statesD(2))$
from *finite-subset[OF this]* **show** *?thesis* **by** *auto*
qed

Collecting all states reachable from target of rules

lemma *finite-ta-rhs-states* [*simp*]:
 $\text{finite } \{q. \exists p. p \mid \in \mid \text{rule-target-states } (\text{rules } \mathcal{A}) \wedge (p = q \vee (p, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid)\}$
(is *finite ?Set*)
proof –
 have $?Set \subseteq \text{fset } (\mathcal{Q } \mathcal{A})$
 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 $\mid \in \mid (\text{rules } \mathcal{A}) \implies \text{length } qs = n \implies (f, n) \mid \in \mid \text{ta-sig } \mathcal{A}$ **unfolding**
ta-sig-def
using *mk-disjoint-finsert* **by fastforce**

lemma *ta-sig-mono*:
 $(\text{rules } \mathcal{A}) \mid \subseteq \mid (\text{rules } \mathcal{B}) \implies \text{ta-sig } \mathcal{A} \mid \subseteq \mid \text{ta-sig } \mathcal{B}$
by (*auto simp: ta-sig-def*)

lemma *finite-eps*:
 $\text{finite } \{q. \exists f ps p. f ps \rightarrow p \mid \in \mid \text{rules } \mathcal{A} \wedge (p = q \vee (p, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid)\}$ (is
finite ?S)
by (*intro finite-subset[OF - finite-ta-rhs-states[of \mathcal{A}]]*) (*auto intro!: bexI*)

lemma *collect-snd-trancl-fset*:
 $\{p. (q, p) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid\} = \text{fset } (\text{snd } \mid^+ \mid (\text{ffilter } (\lambda x. \text{fst } x = q) ((\text{eps } \mathcal{A})^+ \mid)))$
by (*auto simp: image-iff*) *force*

lemma *ta-der-Var*:
 $q \mid \in \mid \text{ta-der } \mathcal{A} (\text{Var } x) \longleftrightarrow x = q \vee (x, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid$
by (*auto simp: collect-snd-trancl-fset*)

lemma *ta-der-Fun*:
 $q \mid \in \mid \text{ta-der } \mathcal{A} (\text{Fun } f ts) \longleftrightarrow (\exists ps p. \text{TA-rule } f ps p \mid \in \mid (\text{rules } \mathcal{A}) \wedge$
 $(p = q \vee (p, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid) \wedge \text{length } ps = \text{length } ts \wedge$
 $(\forall i < \text{length } ts. ps ! i \mid \in \mid \text{ta-der } \mathcal{A} (ts ! i)))$ (is *?Ls \longleftrightarrow ?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*:
 $\text{Var } q \mid \in \mid \text{ta-der}' \mathcal{A} (\text{Var } x) \longleftrightarrow x = q \vee (x, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \mid$
by (*auto simp: collect-snd-trancl-fset*)

lemma *ta-der'-Fun*:
 $Var\ q\ |\in|\ ta\text{-der}'\ \mathcal{A}\ (Fun\ f\ ts) \longleftrightarrow q\ |\in|\ ta\text{-der}\ \mathcal{A}\ (Fun\ f\ ts)$
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*:
 $Fun\ f\ ps\ |\in|\ ta\text{-der}'\ \mathcal{A}\ (Fun\ g\ ts) \longleftrightarrow f = g \wedge length\ ps = length\ ts \wedge (\forall i < length\ ts.\ ps\ !\ i\ |\in|\ ta\text{-der}'\ \mathcal{A}\ (ts\ !\ i))$
proof –
have *f: finite {ss. set ss \subseteq fset (| \cup | (fset-of-list (map (ta-der' \mathcal{A}) ts))) \wedge length ss = length ts}*
by (*intro finite-lists-length-eq auto*)
have *finite {ss. length ss = length ts \wedge ($\forall i < length\ ts.\ ss\ !\ i\ |\in|\ ta\text{-der}'\ \mathcal{A}\ (ts\ !\ i))}$*
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 |\in|\ ta-der \mathcal{A} t*
and *VarI: $\bigwedge q\ v.\ v = q \vee (v, q) |\in|\ (eps\ \mathcal{A})^+ \implies P\ (Var\ v)\ q$*
and *FunI: $\bigwedge f\ ts\ ps\ p\ q.\ f\ ps \rightarrow p |\in|\ rules\ \mathcal{A} \implies length\ ts = length\ ps \implies p = q \vee (p, q) |\in|\ (eps\ \mathcal{A})^+ \implies$*
 $(\bigwedge i.\ i < length\ ts \implies ps\ !\ i\ |\in|\ ta\text{-der}\ \mathcal{A}\ (ts\ !\ i)) \implies$
 $(\bigwedge i.\ i < length\ ts \implies P\ (ts\ !\ i)\ (ps\ !\ i)) \implies P\ (Fun\ f\ ts)\ q$
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 |\in|\ ta-der \mathcal{A} (term-of-gterm t)*
and *Fun: $\bigwedge f\ ts\ ps\ p\ q.\ TA\text{-rule}\ f\ ps\ p\ |\in|\ rules\ \mathcal{A} \implies length\ ts = length\ ps \implies$*
 $p = q \vee (p, q) |\in|\ (eps\ \mathcal{A})^+ \implies$
 $(\bigwedge i.\ i < length\ ts \implies ps\ !\ i\ |\in|\ ta\text{-der}\ \mathcal{A}\ (term\text{-of-gterm}\ (ts\ !\ i))) \implies$
 $(\bigwedge i.\ i < length\ ts \implies P\ (ts\ !\ i)\ (ps\ !\ i)) \implies P\ (GFun\ f\ ts)\ q$
shows *P t q using assms(1)*
by (*induct t arbitrary: q (auto simp: Fun)*)

lemma *ta-der-rule-empty*:
assumes *q |\in|\ ta-der (TA {||} Δ_ϵ) t*
obtains *p where t = Var p p = q \vee (p, q) |\in|\ Δ_ϵ^+*

```

using assms by (cases t) auto

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

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

lemma ta-der-mono:
  (rules A)  $|\subseteq|$  (rules B)  $\implies$  (eps A)  $|\subseteq|$  (eps B)  $\implies$  ta-der A t  $|\subseteq|$  ta-der B t
proof (induct t)
  case (Var x) then show ?case
    by (auto dest: ftrancl-mono[of - eps A eps 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 A)  $|\subseteq|$  (rules B)  $\implies$  (eps A)  $|\subseteq|$  (eps B)  $\implies$  q  $|\in|$  ta-der A t  $\implies$  q  $|\in|$ 
ta-der B t
  using ta-der-mono by blast

lemma ta-der'-ta-der:
  assumes t  $|\in|$  ta-der' A s p  $|\in|$  ta-der A t
  shows p  $|\in|$  ta-der A s using assms
proof (induction arbitrary: p t rule: ta-der'.induct)
  case (2 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  $\forall i < length ts. ss ! i$   $|\in|$  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  $|\in|$  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*:
 $Var\ q\ |\in|\ ta-der'\ \mathcal{A}\ s \implies q\ |\in|\ ta-der\ \mathcal{A}\ s$
using *ta-der'-ta-der* **by** *fastforce*

lemma *ta-der-to-ta-der'*:
 $q\ |\in|\ ta-der\ \mathcal{A}\ s \iff Var\ q\ |\in|\ ta-der'\ \mathcal{A}\ s$
by (*induct s arbitrary: q*) *auto*

lemma *ta-der'-poss*:
assumes $t\ |\in|\ ta-der'\ \mathcal{A}\ s$
shows $poss\ t \subseteq poss\ s$ **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]*: $t\ |\in|\ ta-der'\ \mathcal{A}\ t$
by (*induction t*) *fastforce+*

lemma *ta-der'-eps*:
assumes $Var\ p\ |\in|\ ta-der'\ \mathcal{A}\ s$ **and** $(p, q)\ |\in|\ (eps\ \mathcal{A})^+|$
shows $Var\ q\ |\in|\ ta-der'\ \mathcal{A}\ s$ **using** *assms*
by (*cases s, auto dest: ftrancl-trans*) (*meson ftrancl-trans*)

lemma *ta-der'-trans*:
assumes $t\ |\in|\ ta-der'\ \mathcal{A}\ s$ **and** $u\ |\in|\ ta-der'\ \mathcal{A}\ t$
shows $u\ |\in|\ ta-der'\ \mathcal{A}\ s$ **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 $p: p\ |\in|\ ta-der\ \mathcal{A}\ t\ q\ |\in|\ ta-der\ \mathcal{A}\ C\langle Var\ p\rangle$
shows $q\ |\in|\ ta-der\ \mathcal{A}\ C\langle t\rangle$ **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 A length qs = Suc (length ss + length ts) and
reach: ∀ i < Suc (length ss + length ts). qs ! i |∈| ta-der A ((ss @ C⟨Var p⟩ #
ts) ! i) r = q ∨ (r, q) |∈| (eps A)|⁺|
by *auto*
have *i < Suc (length ss + length ts) ⇒ qs ! i |∈| ta-der A ((ss @ C⟨t⟩ # 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⟨Var q⟩ and (q, q′) |∈| (eps A)|⁺|*
shows *p |∈| ta-der A C⟨Var q⟩*
using *assms* **by** (*meson ta-der-Var ta-der-ctxt*)

lemma *rule-reachable-ctxt-exist:*
assumes *rule: f qs → q |∈| rules A and i < length qs*
shows $\exists C. q |∈| ta-der A (C \langle Var (qs ! i) \rangle)$ **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 A C⟨t⟩*
shows $\exists p. p |∈| ta-der A t \wedge q |∈| ta-der A C⟨Var p⟩$ **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 A length qs = Suc (length ss + length ts) and
reach: ∀ i < Suc (length ss + length ts). qs ! i |∈| ta-der A ((ss @ C⟨t⟩ # ts)
! i)
r = q ∨ (r, q) |∈| (eps A)|⁺|
by *auto*
obtain *p* **where** *p: p |∈| ta-der A t qs ! length ss |∈| ta-der A C⟨Var p⟩*
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 A ((ss @ C⟨Var p⟩ #*
ts) ! i) **for** *i*
using *reach rule(2) p* **by** (*auto simp: p(2) nth-append-Cons*)
then have *q |∈| ta-der A (More f ss C ts)⟨Var p⟩* **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 $\mathcal{A} t \mid\subseteq\mid \mathcal{Q} \mathcal{A} \mid\cup\mid$ *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 $t \implies$ *ta-der* $\mathcal{A} t \mid\subseteq\mid \mathcal{Q} \mathcal{A}$

using *ta-der-states[of $\mathcal{A} t$]* **by** (*auto simp: ground-fvars-term-empty*)

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

lemma *gterm-ta-der-states* [*simp*]:

$q \mid\in\mid$ *ta-der* \mathcal{A} (*term-of-gterm* t) $\implies q \mid\in\mid \mathcal{Q} \mathcal{A}$

by (*intro ground-ta-der-states[THEN fsubsetD, of term-of-gterm t]*) *simp*

lemma *ta-der-states'*:

$q \mid\in\mid$ *ta-der* $\mathcal{A} t \implies q \mid\in\mid \mathcal{Q} \mathcal{A} \implies$ *fvars-term* $t \mid\subseteq\mid \mathcal{Q} \mathcal{A}$

proof (*induct rule: ta-der-induct*)

case (*Fun* $f ts ps p r$)

then have $i < \text{length } ts \implies$ *fvars-term* $(ts ! i) \mid\subseteq\mid \mathcal{Q} \mathcal{A}$ **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*:

$q \mid\in\mid$ *ta-der* $\mathcal{A} t \implies q \notin \mathcal{Q} \mathcal{A} \implies t = \text{Var } q$

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 $t \implies q \mid\in\mid$ *ta-der* $\mathcal{A} t \implies q \mid\in\mid \mathcal{Q} \mathcal{A}$

using *ta-der-not-stateD[of $q \mathcal{A} t$]*

by (*cases t*) *auto*

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

is-Fun $t \implies q \mid\in\mid$ *ta-der* $\mathcal{A} t \implies$ *fvars-term* $t \mid\subseteq\mid \mathcal{Q} \mathcal{A}$

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 $\bigwedge r. r \in | \text{rules } \mathcal{A} \implies r\text{-rhs } r \neq q$
and $\bigwedge e. e \in | \text{eps } \mathcal{A} \implies \text{snd } e \neq q$
shows $q \notin | \text{ta-der } \mathcal{A} (\text{term-of-gterm } t)$ **using** *assms*
by (*cases t*) (*fastforce dest!: assms(1) ftranclD2[of - q]*)

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

lemma *ta-rhs-states-res*: **assumes** *is-Fun t*
shows $\text{ta-der } \mathcal{A} t \subseteq | \text{ta-rhs-states } \mathcal{A}$

proof

fix q **assume** $q \in | \text{ta-der } \mathcal{A} t$
from $\langle \text{is-Fun } t \rangle$ **obtain** *f ts* **where** $t = \text{Fun } f \text{ ts}$ **by** (*cases t, auto*)
from $q \in | \text{ta-der } \mathcal{A} t$ **obtain** $q' \text{ qs}$ **where** $\text{TA-rule } f \text{ qs } q' \in | \text{rules } \mathcal{A}$
and $q = q' \vee (q', q) \in | (\text{eps } \mathcal{A})^+ |$ **by** *auto*
then show $q \in | \text{ta-rhs-states } \mathcal{A}$ **unfolding** *ta-rhs-states-def*
by (*auto intro!: beqI*)

qed

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

lemma *ta-der-adapt-vars-ground* [*simp*]:
 $\text{ground } t \implies \text{ta-der } \mathcal{A} (\text{adapt-vars } t) = \text{ta-der } \mathcal{A} t$
by (*induct t, auto*)

lemma *gterm-of-term-inv'*:
 $\text{ground } t \implies \text{term-of-gterm } (\text{gterm-of-term } t) = \text{adapt-vars } t$
by (*induct t*) (*auto 0 0 intro!: nth-equalityI*)

lemma *map-vars-term-term-of-gterm*:
 $\text{map-vars-term } f (\text{term-of-gterm } t) = \text{term-of-gterm } t$
by (*induct t*) *auto*

lemma *adapt-vars-term-of-gterm*:
 $\text{adapt-vars } (\text{term-of-gterm } t) = \text{term-of-gterm } t$
by (*induct t*) *auto*

lemma *ta-der-term-sig*:
 $q \in | \text{ta-der } \mathcal{A} t \implies \text{ffunas-term } t \subseteq | \text{ta-sig } \mathcal{A}$

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*:

$q \in | \text{ta-der } \mathcal{A} \text{ (term-of-gterm } t) \implies \text{ffunas-gterm } t \subseteq | \text{ta-sig } \mathcal{A}$
using *ta-der-term-sig ffunas-term-of-gterm-conv*
by *fastforce*

ta-lang for terms with arbitrary variable type

lemma *ta-langE*: **assumes** $t \in \text{ta-lang } Q \ \mathcal{A}$
obtains $t' \ q$ **where** $\text{ground } t' \ q \in | \ Q \ q \in | \ \text{ta-der } \mathcal{A} \ t' \ t = \text{adapt-vars } t'$
using *assms unfolding ta-lang-def* **by** *blast*

lemma *ta-langI*: **assumes** $\text{ground } t' \ q \in | \ Q \ q \in | \ \text{ta-der } \mathcal{A} \ t' \ t = \text{adapt-vars } t'$
shows $t \in \text{ta-lang } Q \ \mathcal{A}$
using *assms unfolding ta-lang-def* **by** *blast*

lemma *ta-lang-def2*: $(\text{ta-lang } Q \ (\mathcal{A} :: ('q, 'f) \text{ta}) :: ('f, 'v) \text{terms}) = \{t. \text{ground } t \wedge Q \ | \cap | \ \text{ta-der } \mathcal{A} \ (\text{adapt-vars } t) \neq \{\}\}$
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]*:

$\text{ta-lang } Q \ \mathcal{A} = \text{term-of-gterm } \text{' gta-lang } Q \ \mathcal{A} \ (\text{is } ?Ls = ?Rs)$

proof –

{fix t **assume** $t \in ?Ls$
from *ta-langE*[*OF this*] **obtain** $q \ t'$ **where** $\text{ground } t' \ q \in | \ Q \ q \in | \ \text{ta-der } \mathcal{A} \ t'$
 $t = \text{adapt-vars } t'$
by *blast*
then have $t \in ?Rs$ **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** $t \in ?Rs$ **then have** $t \in ?Ls$
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*:

$\text{term-of-gterm } t \in \text{ta-lang } Q \ \mathcal{A} \longleftrightarrow t \in \text{gta-lang } Q \ \mathcal{A}$

by *(metis (mono-tags, lifting) image-iff ta-lang-to-gta-lang term-of-gterm-inv)*

lemma *gta-lang-def-sym*:

$\text{gterm-of-term } \text{' ta-lang } Q \ \mathcal{A} = \text{gta-lang } Q \ \mathcal{A}$

unfolding *gta-lang-def image-def*

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

lemma *gta-langI [intro]*:

assumes $q \in | \ Q$ **and** $q \in | \ \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } t)$

shows $t \in \text{gta-lang } Q \ \mathcal{A}$ **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 $t \in \text{gta-lang } Q \ \mathcal{A}$
obtains q **where** $q \in Q$ **and** $q \in \text{ta-der } \mathcal{A}$ (*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 $\bigwedge t. \text{ta-der } \mathcal{A} \ t \subseteq \text{ta-der } \mathfrak{B} \ t$ **and** $Q_{\mathcal{A}} \subseteq Q_{\mathfrak{B}}$
shows $\text{gta-lang } Q_{\mathcal{A}} \ \mathcal{A} \subseteq \text{gta-lang } Q_{\mathfrak{B}} \ \mathfrak{B}$
using *assms* **by** (*auto elim!: gta-langE intro!: gta-langI*)

lemma *gta-lang-term-of-gterm* [*simp*]:
term-of-gterm $t \in \text{term-of-gterm } \text{'gta-lang } Q \ \mathcal{A} \longleftrightarrow t \in \text{gta-lang } Q \ \mathcal{A}$
by (*auto elim!: gta-langE intro!: gta-langI*) (*metis term-of-gterm-inv*)

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

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

lemma *ta-syms-lang*: $t \in \text{ta-lang } Q \ \mathcal{A} \implies \text{ffunass-term } t \subseteq \text{ta-sig } \mathcal{A}$
using *gta-lang-subset-rules-funass ffunass-gterm-gterm-of-term ta-der-gterm-sig ta-lang-def2*
by *fastforce*

lemma *gta-lang-Rest-states-conv*:
 $\text{gta-lang } Q \ \mathcal{A} = \text{gta-lang } (Q \ |\cap| \ Q) \ \mathcal{A}$
by (*auto elim!: gta-langE*)

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

Deterministic tree automata

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

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

lemma *ta-detE* [*elim, consumes 1*]: **assumes** *det*: $\text{ta-det } \mathcal{A}$
shows $q \in \text{ta-der } \mathcal{A} \ t \implies q' \in \text{ta-der } \mathcal{A} \ t \implies q = q'$ **using** *assms*

by (induct t arbitrary: q q') (auto simp: ta-det-def, metis nth-equalityI nth-mem)

lemma ta-subset-states: ta-subset $\mathcal{A} \mathcal{B} \implies \mathcal{Q} \mathcal{A} \mid\subseteq\mid \mathcal{Q} \mathcal{B}$
 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 $\mathcal{A} \mathcal{B} \implies$ ta-subset $\mathcal{B} \mathcal{C} \implies$ ta-subset $\mathcal{A} \mathcal{C}$
 unfolding ta-subset-def by auto

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

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

lemma ta-lang-mono': ta-subset $\mathcal{A} \mathcal{B} \implies \mathcal{Q}_{\mathcal{A}} \mid\subseteq\mid \mathcal{Q}_{\mathcal{B}} \implies$ ta-lang $\mathcal{Q}_{\mathcal{A}} \mathcal{A} \subseteq$ ta-lang $\mathcal{Q}_{\mathcal{B}} \mathcal{B}$
 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: $\mathcal{Q} (ta-restrict \mathcal{A} Q) \mid\subseteq\mid \mathcal{Q}$
 by (auto simp: \mathcal{Q} -def ta-restrict-def rule-states-def eps-states-def dest!: fsubsetD)

lemma ta-restrict-states: $\mathcal{Q} (ta-restrict \mathcal{A} Q) \mid\subseteq\mid \mathcal{Q} \mathcal{A}$
 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 - \mathcal{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 $t \mid\subseteq\mid \mathcal{Q} \implies q \mid\in\mid$ ta-der (ta-restrict $\mathcal{A} Q$) $t \implies q \mid\in\mid \mathcal{Q}$
 by (induct t arbitrary: q) (auto simp: ta-restrict-def elim: ftranclE)

lemma ta-derict-ruleI [intro]:

$TA\text{-rule } f\ qs\ q \in \mathcal{A} \implies \text{fset-of-list } qs \subseteq Q \implies q \in Q \implies TA\text{-rule } f\ qs\ q \in \mathcal{A}$
 $q \in \mathcal{A}$ rules (*ta-restrict* \mathcal{A} Q)
by (*auto simp: ta-restrict-def intro!: ta.expand finite-subset[OF - finite-Collect-ta-rule, of - \mathcal{A}]*)

Reachable and productive states: There always is a trim automaton

lemma *finite-ta-reachable* [*simp*]:
 $\text{finite } \{q. \exists t. \text{ground } t \wedge q \in \text{ta-der } \mathcal{A} t\}$
proof –
have $\{q. \exists t. \text{ground } t \wedge q \in \text{ta-der } \mathcal{A} t\} \subseteq \text{fset } (Q\ \mathcal{A})$
using *ground-ta-der-states*[*of - \mathcal{A}]*
by *auto*
from *finite-subset*[*OF this*] **show** *?thesis* **by** *auto*
qed

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

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

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

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

lemma *ta-reachable-gtermI* [*intro*]:
 $q \in \text{ta-der } \mathcal{A}$ (*term-of-gterm* t) $\implies q \in \text{ta-reachable } \mathcal{A}$
by (*intro ta-reachableI*[*of term-of-gterm t*]) *simp*

lemma *ta-reachableI-rule*:
assumes $\text{sub: fset-of-list } qs \subseteq \text{ta-reachable } \mathcal{A}$
and *rule: TA-rule* $f\ qs\ q \in \mathcal{A}$ rules \mathcal{A}
shows $q \in \text{ta-reachable } \mathcal{A}$
 $\exists ts. \text{length } qs = \text{length } ts \wedge (\forall i < \text{length } ts. \text{ground } (ts\ !\ i)) \wedge$
 $(\forall i < \text{length } ts. qs\ !\ i \in \text{ta-der } \mathcal{A} (ts\ !\ i))$ (**is** *?G*)
proof –
{

```

fix  $i$ 
  assume  $i: i < \text{length } qs$ 
  then have  $qs ! i \in | \text{fset-of-list } qs$  by auto
  with  $sub$  have  $qs ! i \in | \text{ta-reachable } \mathcal{A}$  by auto
  from  $\text{ta-reachableE}[OF \text{ this}]$  have  $\exists t. \text{ground } t \wedge qs ! i \in | \text{ta-der } \mathcal{A} t$  by auto
}
then have  $\forall i. \exists t. i < \text{length } qs \longrightarrow \text{ground } t \wedge qs ! i \in | \text{ta-der } \mathcal{A} t$  by auto
from  $\text{choice}[OF \text{ this}]$  obtain  $ts$  where  $ts: \bigwedge i. i < \text{length } qs \implies \text{ground } (ts\ i)$ 
 $\wedge qs ! i \in | \text{ta-der } \mathcal{A} (ts\ i)$  by blast
let  $?t = \text{Fun } f (map\ ts\ [0 .. < \text{length } qs])$ 
have  $gt: \text{ground } ?t$  using  $ts$  by auto
have  $r: q \in | \text{ta-der } \mathcal{A} ?t$  unfolding  $\text{ta-der-Fun}$  using  $rule\ ts$ 
  by  $(intro\ exI[of - qs]\ exI[of - q])\ simp$ 
with  $gt$  show  $q \in | \text{ta-reachable } \mathcal{A}$  by blast
from  $gt\ ts$  show  $?G$  by  $(intro\ exI[of - map\ ts\ [0..<\text{length } qs]])\ simp$ 
qed

```

```

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

```

```

lemma  $\text{ta-reachableI-eps}'$ :
  assumes  $\text{reach}: q \in | \text{ta-reachable } \mathcal{A}$ 
  and  $\text{eps}: (q, q') \in | (\text{eps } \mathcal{A})^+ |$ 
  shows  $q' \in | \text{ta-reachable } \mathcal{A}$ 
proof –
  from  $\text{ta-reachableE}[OF \text{ reach}]$  obtain  $t$  where  $g: \text{ground } t$  and  $\text{res}: q \in | \text{ta-der } \mathcal{A} t$  by auto
  from  $\text{ta-der-trancl-eps}[OF \text{ eps } res]\ g$  show  $?thesis$  by blast
qed

```

```

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

```

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

lemma *finite-ta-productive*:

finite $\{p. \exists q q' C. p = q \wedge q' \in | \text{ta-der } \mathcal{A} C \langle \text{Var } q \rangle \wedge q' \in | P\}$

proof –

{fix $x q C$ **assume** *ass*: $x \notin \text{fset } P$ $q \in | P$ $q \in | \text{ta-der } \mathcal{A} C \langle \text{Var } x \rangle$

then have $x \in \text{fset } (\mathcal{Q} \mathcal{A})$

proof (*cases is-Fun C* $\langle \text{Var } x \rangle$)

case *True*

then show *?thesis* **using** *ta-der-is-fun-fvars-stateD*[*OF - ass*(\exists)]

by *auto*

next

case *False*

then show *?thesis* **using** *ass*

by (*cases C, auto, (metis eps-trancl-statesD)*+))

qed}

then have $\{q \mid q q' C. q' \in | \text{ta-der } \mathcal{A} (C \langle \text{Var } q \rangle) \wedge q' \in | P\} \subseteq \text{fset } (\mathcal{Q} \mathcal{A}) \cup \text{fset } P$ **by** *auto*

from *finite-subset*[*OF this*] **show** *?thesis* **by** *auto*

qed

lemma *ta-productiveE*: **assumes** $q \in | \text{ta-productive } P \mathcal{A}$

obtains $q' C$ **where** $q' \in | \text{ta-der } \mathcal{A} (C \langle \text{Var } q \rangle) q' \in | P$

using *assms*[*unfolded ta-productive-def*] **by** *auto*

lemma *ta-productiveI*:

assumes $q' \in | \text{ta-der } \mathcal{A} (C \langle \text{Var } q \rangle) q' \in | P$

shows $q \in | \text{ta-productive } P \mathcal{A}$

using *assms* **unfolding** *ta-productive-def*

using *finite-ta-productive*

by *auto*

lemma *ta-productiveI'*:

assumes $q \in | \text{ta-der } \mathcal{A} (C \langle \text{Var } p \rangle) q \in | \text{ta-productive } P \mathcal{A}$

shows $p \in | \text{ta-productive } P \mathcal{A}$

using *assms* **unfolding** *ta-productive-def*

by *auto* (*metis (mono-tags, lifting) ctxt-ctxt-compose ta-der-ctxt*)

lemma *ta-productive-setI*:

$q \in | P \implies q \in | \text{ta-productive } P \mathcal{A}$

using *ta-productiveI*[*of q A* \square q]

by *simp*

lemma *ta-reachable-empty-rules* [*simp*]:

rules $\mathcal{A} = \{\|\}$ $\implies \text{ta-reachable } \mathcal{A} = \{\|\}$

by (*auto simp: ta-reachable-def*)

(metis ground.simps(1) ta.exhaust-sel ta-der-rule-empty)

lemma ta-reachable-mono:

ta-subset $\mathcal{A} \mathcal{B} \implies ta\text{-reachable } \mathcal{A} \sqsubseteq ta\text{-reachable } \mathcal{B}$ **using** ta-der-mono'
by (auto simp: ta-reachable-def) blast

lemma ta-reachable-rhs-states:

ta-reachable $\mathcal{A} \sqsubseteq ta\text{-rhs-states } \mathcal{A}$

proof –

{**fix** q **assume** $q \in ta\text{-reachable } \mathcal{A}$
then obtain t **where** ground $t q \in ta\text{-der } \mathcal{A} t$
by (auto simp: ta-reachable-def)
then have $q \in ta\text{-rhs-states } \mathcal{A}$
by (cases t) (auto simp: ta-rhs-states-def intro!: be x I)}
then show ?thesis **by** blast

qed

lemma ta-reachable-eps:

$(p, q) \in (eps \mathcal{A})^{+} \implies p \in ta\text{-reachable } \mathcal{A} \implies (p, q) \in (fRestr (eps \mathcal{A}) (ta\text{-reachable } \mathcal{A}))^{+}$

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 $q \in ta\text{-reachable } \mathcal{A} r \in ta\text{-reachable } \mathcal{A}$

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 $t \sqsubseteq ta\text{-reachable } \mathcal{A}$

shows ta-der $\mathcal{A} t = ta\text{-der } (ta\text{-only-reach } \mathcal{A}) t$ (**is** ?LS = ?RS)

proof –

have ?RS \sqsubseteq ?LS **using** ta-der-mono'[OF ta-restrict-subset]

by fastforce

moreover

{**fix** q **assume** $q \in ?LS$

then have $q \in ?RS$ **using** assms

proof (induct rule: ta-der-induct)

case (Fun $f ts ps p q$)

from Fun(2, 6) **have** ta-reach [simp]: $i < length ps \implies fvars\text{-term } (ts ! i)$

$\sqsubseteq ta\text{-reachable } \mathcal{A}$ **for** i

by auto (metis ffUnionI fimage-fset fnth-mem funionI2 length-map nth-map sup.orderE)

from Fun **have** $r: i < length ts \implies ps ! i \in ta\text{-der } (ta\text{-only-reach } \mathcal{A}) (ts !$

i)
 $i < \text{length } ts \implies ps ! i \in | \text{ta-reachable } \mathcal{A} \text{ for } i$
by (*auto*) (*metis ta-reach ta-der-ta-derict-states*) +
then have $f ps \rightarrow p \in | \text{rules } (\text{ta-only-reach } \mathcal{A})$
using *Fun(1, 2)*
by (*intro ta-derict-ruleI*)
 $(\text{fastforce simp: in-fset-conv-nth intro!: ta-reachableI-rule}[OF - \text{Fun}(1)]) +$
then show $?case \text{ using } \text{ta-reachable-eps}[of p q] \text{ta-reachableI-rule}[OF - \text{Fun}(1)]$
 $r \text{Fun}(2, 3)$
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*:
 $\text{ta-der } \mathcal{A} (\text{term-of-gterm } t) = \text{ta-der } (\text{ta-only-reach } \mathcal{A}) (\text{term-of-gterm } t)$
using *ta-der-only-reach[of term-of-gterm t A]*
by *simp*

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

lemma *ta-only-reach-reachable*:
 $Q (\text{ta-only-reach } \mathcal{A}) \subseteq | \text{ta-reachable } (\text{ta-only-reach } \mathcal{A})$
using *ta-restrict-states-Q[of A ta-reachable A]*
by *auto*

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

lemma *L-only-reach*: $\mathcal{L} (\text{reg-reach } R) = \mathcal{L} R$
using *gta-only-reach-lang*
by (*auto simp: 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*:
 $(p, q) \in | (eps\ \mathcal{A}) |^+ \implies q \in | ta-productive\ P\ \mathcal{A} \implies p \in | ta-productive\ P\ \mathcal{A}$
using *ta-der-ctxt[of q \mathcal{A} \square \langle Var\ p \rangle]*
by (*auto simp: ta-productive-def ta-der-trancl-eps*)

lemma *ta-only-prod-eps*:
 $(p, q) \in | (eps\ \mathcal{A}) |^+ \implies q \in | ta-productive\ P\ \mathcal{A} \implies (p, q) \in | (eps\ (ta-only-prod\ P\ \mathcal{A})) |^+$
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*:
 $q \in | ta-der\ \mathcal{A}\ t \implies q \in | ta-productive\ P\ \mathcal{A} \implies q \in | ta-der\ (ta-only-prod\ P\ \mathcal{A})\ t$
proof (*induct rule: ta-der-induct*)
case (*Fun f ts ps p q*)
let $?A = ta-only-prod\ P\ \mathcal{A}$
have $pr: p \in | ta-productive\ P\ \mathcal{A}\ i < length\ ts \implies ps\ !\ i \in | ta-productive\ P\ \mathcal{A}$
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 $f\ ps \rightarrow p \in | rules\ ?A$ **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*:
 $q \in | ta-der\ (ta-only-prod\ P\ \mathcal{A})\ t \implies q \in | ta-der\ \mathcal{A}\ t$
by (*meson ta-der-el-mono ta-restrict-subset ta-subset-def*)

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

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

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

lemma *ta-productive-ta-only-prod [simp]*:
 $ta\text{-}productive\ P\ (ta\text{-}only\text{-}prod\ P\ \mathcal{A}) = ta\text{-}productive\ P\ \mathcal{A}$ (is $?LS = ?RS$)
proof –
 have $?LS \subseteq | ?RS$ **using** $ta\text{-}der\text{-}mono'[OF\ ta\text{-}restrict\text{-}subset]$
 using $finite\text{-}ta\text{-}productive[of\ \mathcal{A}\ P]$
 by (*auto simp: ta-productive-def*) *fastforce*
moreover have $?RS \subseteq | ?LS$ **using** $ta\text{-}der\text{-}only\text{-}prod$
 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*:
 $Q\ (ta\text{-}only\text{-}prod\ P\ \mathcal{A}) \subseteq | ta\text{-}productive\ P\ (ta\text{-}only\text{-}prod\ P\ \mathcal{A})$
 using $ta\text{-}restrict\text{-}states\text{-}Q$ **by** *force*

lemma *ta-only-prod-reachable*:
assumes $all\text{-}reach: Q\ \mathcal{A} \subseteq | ta\text{-}reachable\ \mathcal{A}$
shows $Q\ (ta\text{-}only\text{-}prod\ P\ \mathcal{A}) \subseteq | ta\text{-}reachable\ (ta\text{-}only\text{-}prod\ P\ \mathcal{A})$ (is $?Ls \subseteq | ?Rs$)
proof –
 {**fix** q **assume** $q \in | ?Ls$
then obtain t **where** $ground\ t\ q \in | ta\text{-}der\ \mathcal{A}\ t\ q \in | ta\text{-}productive\ P\ \mathcal{A}$
 using $fsubsetD[OF\ ta\text{-}only\text{-}prod\text{-}productive[of\ \mathcal{A}\ P]]$

```

using fsubsetD[OF fsubset-trans[OF ta-restrict-states all-reach, of ta-productive
P A]]
  by (auto elim!: ta-reachableE)
  then have  $q \in | ?Rs$ 
    by (intro ta-reachableI[where  $?A = ta-only-prod P A$  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 A)) A
  by (rule ta-subset-trans, (rule ta-restrict-subset)+)

```

```

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

```

```

lemma ta-productive-aux:
  assumes  $\mathcal{Q} A \subseteq | ta-reachable A q \in | ta-der A (C\langle t \rangle)$ 
  shows  $\exists C'. ground-ctxt C' \wedge q \in | ta-der A (C'\langle t \rangle)$  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  $q': f qs \rightarrow q' \in | rules A q' = q \vee (q', q) \in |$ 
  (eps A)+
   $qs ! length ts1 \in | ta-der A (C\langle t \rangle) length qs = Suc (length ts1 + length ts2)$ 
  by simp (metis less-add-Suc1 nth-append-length)
  { fix  $i$  assume  $i < length qs$ 
    then have  $qs ! i \in | \mathcal{Q} A$  using  $q'(1)$ 
    by (auto dest!: rule-statesD(4))
    then have  $\exists t. ground t \wedge qs ! i \in | ta-der A t$  using assms(1)
    by (simp add: ta-reachable-def) force}
  then obtain  $ts$  where  $ts: i < length qs \implies ground (ts i) \wedge qs ! i \in | ta-der A$ 
  ( $ts i$ ) for  $i$  by metis
  obtain  $C'$  where  $C: ground-ctxt C' qs ! length ts1 \in | ta-der A C'\langle t \rangle$  using
  More(1)[OF q'(3)] by blast
  define  $D$  where  $D \equiv More f (map ts [0..<length ts1]) C' (map ts [Suc (length$ 
   $ts1)..<Suc (length ts1 + length ts2)])$ 
  have ground-ctxt D unfolding D-def using ts C(1) q'(4) by auto
  moreover have  $q \in | ta-der A D\langle t \rangle$  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  $\mathcal{Q} A \subseteq | ta-reachable A$ 

```

shows $ta\text{-productive } Q \mathcal{A} = \{ | q | q q' C. \text{ground-ctxt } C \wedge q' | \in | ta\text{-der } \mathcal{A} (C \langle \text{Var } q \rangle) \wedge q' | \in | Q | \}$
using $ta\text{-productive-aux}[OF \text{ assms}]$
by ($auto \text{ simp: } ta\text{-productive-def intro!: finite-subset}[OF - \text{finite-ta-productive, of } - \mathcal{A} Q]) \text{ force+}$

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

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

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

lemma $reg\text{-is-trim-trim-reg [simp]: reg-is-trim } (trim\text{-reg } R)$
unfolding $reg\text{-is-trim-def trim-reg-def}$
by (simp add: trim-ta)

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

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

lemmas $obtain\text{-trimmed-ta} = trim\text{-ta trim-gta-lang ta-subset-det}[OF trim\text{-ta-subset}]$

lemma $\mathcal{L}\text{-trim-ta-sig:}$
assumes $reg\text{-is-trim } R \mathcal{L} R \subseteq \mathcal{T}_G (fset \mathcal{F})$
shows $ta\text{-sig } (ta R) | \subseteq | \mathcal{F}$
proof –
{fix } r **assume** $r: r | \in | rules (ta R)$
then obtain $f ps p$ **where** $[simp]: r = f ps \rightarrow p$ **by** ($\text{cases } r$) **auto**
from $r \text{ assms}(1)$ **have** $fset\text{-of-list } ps | \subseteq | ta\text{-reachable } (ta R)$
by ($auto \text{ simp add: rule-statesD}(4) \text{ reg-is-trim-def ta-is-trim-def}$)
from $ta\text{-reachableI-rule}[OF \text{ this, of } f p] r$
obtain ts **where** $ts: length \ ts = length \ ps \ \forall \ i < \ length \ ps. \ ground \ (ts \ ! \ i)$
 $\ \forall \ i < \ length \ ps. \ ps \ ! \ i | \in | ta\text{-der } (ta R) (ts \ ! \ i)$
by $auto$
obtain $C q$ **where** $ctxt: \text{ground-ctxt } C q | \in | ta\text{-der } (ta R) (C \langle \text{Var } p \rangle) q | \in | fin$
 R

```

    using assms(1) unfolding reg-is-trim-def
  by (metis ⟨r = f ps → p⟩ fsubsetI r rule-statesD(2) ta-productiveE ta-productive-aux
ta-is-trim-def)
  from ts ctxt r have reach: q |∈| ta-der (ta R) C⟨Fun f ts⟩
    by auto (metis ta-der-Fun ta-der-ctxt)
  have gr: ground C⟨Fun f ts⟩ using ts(1, 2) ctxt(1)
    by (auto simp: in-set-conv-nth)
  then have C⟨Fun f ts⟩ ∈ ta-lang (fin R) (ta R) using ctxt(1, 3) ts(1, 2)
    apply (intro ta-langI[OF - - reach, of fin R C⟨Fun f ts⟩])
    apply (auto simp del: adapt-vars-ctxt)
    by (metis gr adapt-vars2 adapt-vars-adapt-vars)
  then have *: gterm-of-term C⟨Fun f ts⟩ ∈  $\mathcal{L}$  R using gr
    by (auto simp:  $\mathcal{L}$ -def)
  then have funas-gterm (gterm-of-term C⟨Fun f ts⟩) ⊆ fset  $\mathcal{F}$  using assms(2)
gr
    by (auto simp:  $\mathcal{T}_G$ -equivalent-def)
  moreover have (f, length ps) ∈ funas-gterm (gterm-of-term C⟨Fun f ts⟩)
    using ts(1) by (auto simp: funas-gterm-gterm-of-term[OF gr])
  ultimately have (r-root r, length (r-lhs-states r)) |∈|  $\mathcal{F}$ 
    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 |!  $\Delta$  = { | TA-rule (g h) (map f qs) (f q) | h qs q. TA-rule h qs q
  |∈|  $\Delta$  }
  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 → q | f qs q. f qs → q |∈| 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 $(h\ p, q) \in |(\text{map-both } h\ |^{\dagger}\ \text{eps } \mathcal{A})|^{+}| \text{finj-on } h\ (\mathcal{Q}\ \mathcal{A})\ p \in | \mathcal{Q}\ \mathcal{A}$

obtains *q'* where $q = h\ q'\ (p, q') \in |(\text{eps } \mathcal{A})|^{+}| q' \in | \mathcal{Q}\ \mathcal{A}$

using *assms(1)[unfolded ftrancl-map-both-fsubset[OF assms(2), of eps \mathcal{A} , simplified]]*

using $\langle \text{finj-on } h\ (\mathcal{Q}\ \mathcal{A}) \rangle$ [*unfolded finj-on-def', rule-format, OF $\langle p \in | \mathcal{Q}\ \mathcal{A} \rangle$*]

by (*metis Pair-inject eps-trancl-statesD prod-fun-fimageE*)

lemma *ta-der-fmap-states-inv-superset*:

assumes $\mathcal{Q}\ \mathcal{A} \subseteq | \mathcal{B}$ *finj-on* $h\ \mathcal{B}$

and $q \in | \text{ta-der } (\text{fmap-states-ta } h\ \mathcal{A})\ (\text{term-of-gterm } t)$

shows *the-finv-into* $\mathcal{B}\ h\ q \in | \text{ta-der } \mathcal{A}\ (\text{term-of-gterm } t)$ 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 $x \in | \mathcal{Q}\ \mathcal{A} \implies \text{the-finv-into } (\mathcal{Q}\ \mathcal{A})\ h\ (h\ x) = \text{the-finv-into } \mathcal{B}\ h\ (h\ x)$ 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* $h\ (\mathcal{Q}\ \mathcal{A})\ q \in | \text{ta-der } (\text{fmap-states-ta } h\ \mathcal{A})\ (\text{term-of-gterm } t)$

shows *the-finv-into* $(\mathcal{Q}\ \mathcal{A})\ h\ q \in | \text{ta-der } \mathcal{A}\ (\text{term-of-gterm } t)$

by (*simp add: ta-der-fmap-states-inv-superset assms*)

lemma *ta-der-to-fmap-states-der*:

assumes $q \in | \text{ta-der } \mathcal{A}\ (\text{term-of-gterm } t)$

shows $h\ q \in | \text{ta-der } (\text{fmap-states-ta } h\ \mathcal{A})\ (\text{term-of-gterm } t)$ 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*) = $h\ |^{\dagger}| \text{ta-der } \mathcal{A}\ (\text{term-of-gterm } t)$

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* \in | *rules* \mathcal{A} *TA-rule g ps q* \in | *rules* \mathcal{A}
then have *TA-rule g* (*map f ps*) (*f p*) \in | *rules* (*fmap-states-ta f* \mathcal{A})
TA-rule g (*map f ps*) (*f q*) \in | *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 $?Ls \implies ?Rs$
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* \in | *rules* \mathcal{A} *TA-rule g qs q* \in | *rules* \mathcal{A}
then have *map f ps* = *map f qs* $\implies ps = qs$ **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 $?Rs \implies ?Ls$ **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 ($\mathcal{Q} \mathcal{A}$) $\implies \mathcal{Q} \subseteq \mathcal{Q} \mathcal{A} \implies$ *gta-lang* (*f* | \uparrow \mathcal{Q}) (*fmap-states-ta f* \mathcal{A}) =
gta-lang $\mathcal{Q} \mathcal{A}$
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 ($\mathcal{Q} \mathcal{A} \cup \mathcal{Q}$) \implies *gta-lang* (*f* | \uparrow \mathcal{Q}) (*fmap-states-ta f* \mathcal{A}) = *gta-lang* $\mathcal{Q} \mathcal{A}$
using *ta-der-fmap-states-conv*[*OF fsubset-finj-on*[*of f* $\mathcal{Q} \mathcal{A} \cup \mathcal{Q} \mathcal{Q} \mathcal{A}$]]
by (*auto simp*: *finj-on-def' elim!*: *gta-langE intro!*: *gta-langI*) *fastforce*

definition *funs-ta* :: ($'q, 'f$) *ta* $\implies 'f$ *fset* **where**
funs-ta $\mathcal{A} = \{f \mid f \text{ qs } q. \text{ TA-rule } f \text{ qs } q \in \text{rules } \mathcal{A}\}$

lemma *funs-ta*[*code*]:
funs-ta $\mathcal{A} = (\lambda r. \text{case } r \text{ of TA-rule } f \text{ ps } p \implies f) \mid \uparrow (\text{rules } \mathcal{A})$ (**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* $\{f. \exists \text{qs } q. \text{ TA-rule } f \text{ qs } q \in \text{rules } \mathcal{A}\}$ *fset* $?Rs$])

lemma *finite-funs-ta* [*simp*]:

finite $\{f. \exists qs q. TA\text{-rule } f \text{ } qs \text{ } q \mid \in \mid \text{rules } \mathcal{A}\}$
by (*intro finite-subset*[of $\{f. \exists qs q. TA\text{-rule } f \text{ } qs \text{ } q \mid \in \mid \text{rules } \mathcal{A}\}$ *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 $f \mid \in \mid \text{funs-ta } \mathcal{A}$
obtains $ps \text{ } p$ **where** $TA\text{-rule } f \text{ } ps \text{ } p \mid \in \mid \text{rules } \mathcal{A}$ **using** *assms*
by (*auto simp: funs-ta-def*)

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

lemma *fmap-funs-ta-cong*:
 $(\bigwedge x. x \mid \in \mid \text{funs-ta } \mathcal{A} \implies h \text{ } x = k \text{ } x) \implies \mathcal{A} = \mathcal{B} \implies \text{fmap-funs-ta } h \text{ } \mathcal{A} =$
 $\text{fmap-funs-ta } k \text{ } \mathcal{B}$
by (*force simp: fmap-funs-ta-def'*)

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

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

lemma *fmap-states-ta-id* [*simp*]:
 $\text{fmap-states-ta } id \text{ } \mathcal{A} = \mathcal{A}$
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*:
 $\text{fmap-funs-ta } h \text{ } (\text{fmap-funs-ta } k \text{ } A) = \text{fmap-funs-ta } (h \circ k) \text{ } A$
proof –
have $r \mid \in \mid \text{rules } A \implies \text{map-ta-rule } id \text{ } h \text{ } (\text{map-ta-rule } id \text{ } k \text{ } r) = \text{map-ta-rule } id$
 $(\lambda x. h \text{ } (k \text{ } x)) \text{ } r$ **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*:
 $\text{fmap-funs-reg } h \text{ } (\text{fmap-funs-reg } k \text{ } A) = \text{fmap-funs-reg } (h \circ k) \text{ } A$
using *fmap-funs-ta-comp* **unfolding** *fmap-funs-reg-def*
by *auto*

lemma *fmap-states-ta-comp*:
 $\text{fmap-states-ta } h \text{ } (\text{fmap-states-ta } k \text{ } A) = \text{fmap-states-ta } (h \circ k) \text{ } A$
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* | \uparrow | *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*:
 $q \in |ta-der\ A\ t| \implies |ffuns-term\ t| \subseteq |funs-ta\ A|$
proof (*induct t arbitrary: q*)
case (*Fun f ts*)
then have $f \in |funs-ta\ A|$ **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*:
 $q \in |ta-der\ A\ t| \implies q \in |ta-der\ (fmap-funs-ta\ f\ A)\ (map-funs-term\ f\ t)|$
by (*induct t arbitrary: q*) (*auto 0 4 simp: fmap-funs-ta-def'*)

lemma *ta-der-fmap-states-ta*:
assumes $q \in |ta-der\ A\ t|$
shows $h\ q \in |ta-der\ (fmap-states-ta\ h\ A)\ (map-vars-term\ h\ t)|$
proof –
have [*intro*]: $(q, q') \in |(eps\ A)|^+ \implies (h\ q, h\ q') \in |(eps\ (fmap-states-ta\ h\ A))|^+$
for $q\ q'$
by (*force intro!: francl-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 $f\ (map\ h\ ps) \rightarrow h\ p \in |rules\ (fmap-states-ta\ h\ A)|$
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 $f \uparrow |ta-der\ A\ (term-of-gterm\ s)| \subseteq |ta-der\ (fmap-states-ta\ f\ A)\ (term-of-gterm\ s)|$
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 $ta-der\ (fmap-states-ta\ f\ A)\ (term-of-gterm\ s) \subseteq |f \uparrow |ta-der\ A\ (term-of-gterm\ s)|$
using *ta-der-fmap-states-conv[OF assms]* **by** *auto*

lemma *fmap-funs-ta-der'*:
 $q \in |ta-der\ (fmap-funs-ta\ h\ A)\ t| \implies \exists t'. q \in |ta-der\ A\ t' \wedge map-funs-term\ h\ t' = t|$

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**
 $\bigwedge i. i < \text{length } ts \implies ps ! i |∈| \text{ta-der } A (ts' i) \wedge \text{map-funs-term } h (ts' i) = ts$
! 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 \implies 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*:
 $\text{gta-lang } Q (\text{fmap-funs-ta } h \mathcal{A}) = \text{map-gterm } h \text{ ' gta-lang } Q \mathcal{A}$ (**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 empty-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-L*:
 $\mathcal{L} (\text{fmap-funs-reg } h R) = \text{map-gterm } h \text{ ' } \mathcal{L} R$
using *fmap-funs-gta-lang[of fin R h]*
by (*auto simp: fmap-funs-reg-def L-def*)

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

lemma *ta-reachable-fmap-funs-ta* [*simp*]:
 $\text{ta-reachable } (\text{fmap-funs-ta } f A) = \text{ta-reachable } A$ **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*:

$fin (reg-Restr-Q_f R) \mid\subseteq\mid \mathcal{Q}_r (reg-Restr-Q_f R)$
by (*auto simp: reg-Restr-Q_f-def*)

lemma *fmap-states-reg-Restr-Q_f-fin*:
 $finj-on f (\mathcal{Q} \mathcal{A}) \implies fin (fmap-states-reg f (reg-Restr-Q_f R)) \mid\subseteq\mid \mathcal{Q}_r (fmap-states-reg f (reg-Restr-Q_f R))$
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*:
 $finite \{f qs \rightarrow (a, b) \mid f qs a b. f map fst qs \rightarrow a \mid\in\mid rules \mathcal{A} \wedge f map snd qs \rightarrow b \mid\in\mid rules \mathfrak{B}\}$ (*is finite ?set*)
proof –
have $?set \subseteq (\lambda (ra, rb). case ra of f ps \rightarrow p \implies case rb of g qs \rightarrow q \implies f (zip ps qs) \rightarrow (p, q))$ ‘(*srules* $\mathcal{A} \times$ *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* $\mathcal{A} \times$ *srules* \mathfrak{B} , *THEN finite-subset*[*OF this*]]
show *?thesis* **by** (*auto simp: srules-def*)
qed

— The product automaton of the automata \mathcal{A} and \mathcal{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} \mid\times\mid \mathcal{Q} \mathcal{B}) \mid\times\mid \mathcal{Q} \mathcal{A} \mid\times\mid \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} \mid\times\mid \mathcal{Q} \mathcal{B}) \mid\times\mid \mathcal{Q} \mathcal{A} \mid\times\mid \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*]: $f qs \rightarrow q \mid\in\mid rules (prod-ta \mathcal{A} \mathcal{B}) \iff f qs \rightarrow q \mid\in\mid prod-ta-rules \mathcal{A} \mathcal{B}$
 $r \mid\in\mid rules (prod-ta \mathcal{A} \mathcal{B}) \iff r \mid\in\mid prod-ta-rules \mathcal{A} \mathcal{B}$
by (*auto simp: prod-ta-def*)

lemma *prod-ta-states*:

$\mathcal{Q} (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \mid \subseteq \mid \mathcal{Q} \ \mathcal{A} \ \mid \times \mid \mathcal{Q} \ \mathcal{B}$

proof –

{**fix** q **assume** $q \mid \in \mid \text{rule-states } (\text{rules } (\text{prod-ta } \mathcal{A} \ \mathcal{B}))$

then obtain $f \ ps \ p$ **where** $f \ ps \rightarrow p \mid \in \mid \text{rules } (\text{prod-ta } \mathcal{A} \ \mathcal{B})$ **and** $q \mid \in \mid \text{fset-of-list}$

$ps \vee p = q$

by (*metis rule-statesE*)

then have $f \ st \ q \mid \in \mid \mathcal{Q} \ \mathcal{A} \ \wedge \ \text{snd } q \mid \in \mid \mathcal{Q} \ \mathcal{B}$

using *rule-statesD(2, 4)[of f map fst ps fst p A]*

using *rule-statesD(2, 4)[of f map snd ps snd p B]*

by *auto*}

moreover

{**fix** q **assume** $q \mid \in \mid \text{eps-states } (\text{eps } (\text{prod-ta } \mathcal{A} \ \mathcal{B}))$ **then have** $f \ st \ q \mid \in \mid \mathcal{Q} \ \mathcal{A} \ \wedge$
 $\text{snd } q \mid \in \mid \mathcal{Q} \ \mathcal{B}$

by (*auto simp: eps-states-def prod-ta-def prod-eps-def dest: eps-statesD*)}

ultimately show *?thesis*

by (*auto simp: Q-def*) *blast+*

qed

lemma *prod-ta-det*:

assumes *ta-det A* **and** *ta-det B*

shows *ta-det (prod-ta A B)*

using *assms unfolding ta-det-def prod-ta-def prod-eps-def*

by *auto*

lemma *prod-ta-sig*:

$\text{ta-sig } (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \mid \subseteq \mid \text{ta-sig } \mathcal{A} \ \mid \cup \mid \text{ta-sig } \mathcal{B}$

proof (*rule fsubsetI*)

fix x

assume $x \mid \in \mid \text{ta-sig } (\text{prod-ta } \mathcal{A} \ \mathcal{B})$

hence $x \mid \in \mid \text{ta-sig } \mathcal{A} \ \vee \ x \mid \in \mid \text{ta-sig } \mathcal{B}$

unfolding *ta-sig-def prod-ta-def*

using *image-iff* **by** *fastforce*

thus $x \mid \in \mid \text{ta-sig } (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \implies x \mid \in \mid \text{ta-sig } \mathcal{A} \ \mid \cup \mid \text{ta-sig } \mathcal{B}$

by *simp*

qed

lemma *from-prod-eps*:

$(p, q) \mid \in \mid (\text{eps } (\text{prod-ta } \mathcal{A} \ \mathcal{B})) \mid^+ \implies (\text{snd } p, \text{snd } q) \mid \notin \mid (\text{eps } \mathcal{B}) \mid^+ \implies \text{snd } p =$
 $\text{snd } q \ \wedge \ (fst \ p, fst \ q) \mid \in \mid (\text{eps } \mathcal{A}) \mid^+$

$(p, q) \mid \in \mid (\text{eps } (\text{prod-ta } \mathcal{A} \ \mathcal{B})) \mid^+ \implies (fst \ p, fst \ q) \mid \notin \mid (\text{eps } \mathcal{A}) \mid^+ \implies fst \ p = fst$
 $q \ \wedge \ (\text{snd } p, \text{snd } q) \mid \in \mid (\text{eps } \mathcal{B}) \mid^+$

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-epsA*:

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

lemma *to-prod-epsB*:

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

lemma *to-prod-eps*:

$(p, q) \in | (eps \mathcal{A}) |^+ \implies (p', q') \in | (eps \mathcal{B}) |^+ \implies ((p, p'), (q, q')) \in | (eps (prod-ta \mathcal{A} \mathcal{B})) |^+$

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 $((a, c), b, c) \in | (eps (prod-ta \mathcal{A} \mathcal{B})) |^+ \text{ using } finite-prod-eps$

by (*auto simp: prod-ta-def prod-eps-def dest: eps-statesD intro!: fr-into-trancl ftrancl-into-trancl*)

moreover have $((b, c), b, d) \in | (eps (prod-ta \mathcal{A} \mathcal{B})) |^+ \text{ using } finite-prod-eps$
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 $((b, q), b, r) \in | (eps (prod-ta \mathcal{A} \mathcal{B})) |^+ \text{ using } finite-prod-eps$

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 $((b, q'), (c, q')) \in | (eps (prod-ta \mathcal{A} \mathcal{B})) |^+$

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-A-B-der1*:

assumes $q \in | ta-der (prod-ta \mathcal{A} \mathcal{B}) (term-of-gterm t)$

shows $fst q \in | ta-der \mathcal{A} (term-of-gterm t) \text{ using } assms$

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-A-B-der2:*

assumes $q \in | \text{ta-der } (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \ (\text{term-of-gterm } t)$
shows $\text{snd } q \in | \text{ta-der } \mathcal{B} \ (\text{term-of-gterm } t)$ **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 *A-B-der-to-prod-ta:*

assumes $\text{fst } q \in | \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } t)$ $\text{snd } q \in | \text{ta-der } \mathcal{B} \ (\text{term-of-gterm } t)$
shows $q \in | \text{ta-der } (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \ (\text{term-of-gterm } t)$ **using** *assms*
proof (*induct t arbitrary: q*)
case (*GFun f ts*)
from *GFun(2, 3)* **obtain** $ps \ qs \ p \ q'$ **where**
rules: f ps \rightarrow p $\in | \text{rules } \mathcal{A} \ f \ qs \rightarrow q' \in | \text{rules } \mathcal{B} \ \text{length } ps = \text{length } ts \ \text{length } ps = \text{length } qs$ and
eps: p = fst q \vee (p, fst q) $\in | (\text{eps } \mathcal{A})|^{+}$ q' = snd q \vee (q', snd q) $\in | (\text{eps } \mathcal{B})|^{+}$
and
steps: $\forall i < \text{length } qs. ps ! i \in | \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } (ts ! i))$
 $\forall i < \text{length } qs. qs ! i \in | \text{ta-der } \mathcal{B} \ (\text{term-of-gterm } (ts ! i))$
by *auto*
from *rules* **have** $st: p \in | \mathcal{Q} \ \mathcal{A} \ q' \in | \mathcal{Q} \ \mathcal{B}$ **by** (*auto dest: rule-statesD*)
have $(p, \text{snd } q) = q \vee ((p, q'), q) \in | (\text{eps } (\text{prod-ta } \mathcal{A} \ \mathcal{B}))|^{+}$ **using** *eps st*
using *to-prod-epsB[of q' snd q B fst q A]*
using *to-prod-epsA[of p fst q A snd q B]*
using *to-prod-eps[of p fst q A q' snd q 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-epsB intro!: exI[of - p] exI[of - q'] exI[of - zip ps qs])
qed

lemma *prod-ta-der:*

$q \in | \text{ta-der } (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \ (\text{term-of-gterm } t) \iff$
 $\text{fst } q \in | \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } t) \wedge \text{snd } q \in | \text{ta-der } \mathcal{B} \ (\text{term-of-gterm } t)$
using *prod-ta-der-to-A-B-der1 prod-ta-der-to-A-B-der2 A-B-der-to-prod-ta*
by *blast*

lemma *intersect-ta-gta-lang:*

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

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

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

— Union of tree automata

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

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

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

lemma *ta-union-eps-disj-states*:
assumes $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\}$ **and** $(p, q) \in | (\text{eps } (\text{ta-union } \mathcal{A} \mathcal{B}))|^{+}$
shows $(p, q) \in | (\text{eps } \mathcal{A})|^{+} \vee (p, q) \in | (\text{eps } \mathcal{B})|^{+}$ **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*]:
 $(p, q) \in | (\text{eps } \mathcal{A})|^{+} \implies (p, q) \in | (\text{eps } (\text{ta-union } \mathcal{A} \mathcal{B}))|^{+}$
 $(p, q) \in | (\text{eps } \mathcal{B})|^{+} \implies (p, q) \in | (\text{eps } (\text{ta-union } \mathcal{A} \mathcal{B}))|^{+}$
by (*auto simp add: in-ftrancl-UnI ta-union-def*)

lemma *disj-states-eps* [*simp*]:
 $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\} \implies f \text{ ps } \rightarrow p \in | \text{rules } \mathcal{A} \implies (p, q) \in | (\text{eps } \mathcal{B})|^{+} \longleftrightarrow \text{False}$
 $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\} \implies f \text{ ps } \rightarrow p \in | \text{rules } \mathcal{B} \implies (p, q) \in | (\text{eps } \mathcal{A})|^{+} \longleftrightarrow \text{False}$
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** $q \in | \text{ta-der } (\text{ta-union } \mathcal{A} \mathcal{B}) t$
shows $q \in | \text{ta-der } \mathcal{A} t \vee q \in | \text{ta-der } \mathcal{B} t$ **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 $dist: fset-of-list\ ps\ |\subseteq|\ \mathcal{Q}\ \mathcal{A} \implies i < length\ ts \implies ps\ !\ i\ |\in|\ ta-der\ \mathcal{A}\ (ts\ !\ i)$
 $fset-of-list\ ps\ |\subseteq|\ \mathcal{Q}\ \mathcal{B} \implies i < length\ ts \implies ps\ !\ i\ |\in|\ ta-der\ \mathcal{B}\ (ts\ !\ i)$ **for** i
using $Fun(2)\ Fun(5)[of\ i]\ assms(1)$
by ($auto\ dest!: ta-der-not-stateD\ fsubsetD$)
from $Fun(1)$ **consider** ($a\ fset-of-list\ ps\ |\subseteq|\ \mathcal{Q}\ \mathcal{A} \mid b\ fset-of-list\ ps\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$)
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** $Q_A\ |\subseteq|\ \mathcal{Q}\ \mathcal{A}$ **and** $Q_B\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$
shows $gta-lang\ (Q_A\ |\cup|\ Q_B)\ (ta-union\ \mathcal{A}\ \mathcal{B}) = gta-lang\ Q_A\ \mathcal{A}\ \cup\ gta-lang\ Q_B\ \mathcal{B}$
(is $?Ls = ?Rs$ **)**
proof –
{fix s **assume** $s \in ?Ls$ **then obtain** q
where $w: q\ |\in|\ Q_A\ |\cup|\ Q_B\ q\ |\in|\ ta-der\ (ta-union\ \mathcal{A}\ \mathcal{B})\ (term-of-gterm\ s)$
by ($auto\ elim: gta-langE$)
from $ta-union-der-disj-states[OF\ assms(1)\ w(2)]$ **consider**
 $(a)\ q\ |\in|\ ta-der\ \mathcal{A}\ (term-of-gterm\ s) \mid q\ |\in|\ ta-der\ \mathcal{B}\ (term-of-gterm\ s)$ **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: $\mathcal{L}\ (reg-union\ R\ L) = \mathcal{L}\ R\ \cup\ \mathcal{L}\ L$

proof –
let $?inl = Inl :: 'b \Rightarrow 'b + 'c$ **let** $?inr = Inr :: 'c \Rightarrow 'b + 'c$
let $?fr = ?inl\ |\uparrow\ (fin\ R\ |\cap|\ \mathcal{Q}_r\ R)$ **let** $?fl = ?inr\ |\uparrow\ (fin\ L\ |\cap|\ \mathcal{Q}_r\ L)$
have $[simp]: gta-lang\ (?fr\ |\cup|\ ?fl)\ (ta-union\ (fmap-states-ta\ ?inl\ (ta\ R))\ (fmap-states-ta\ ?inr\ (ta\ L))) =$
 $gta-lang\ ?fr\ (fmap-states-ta\ ?inl\ (ta\ R))\ \cup\ gta-lang\ ?fl\ (fmap-states-ta\ ?inr\ (ta\ L))$
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 |∩| Qr
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 |∩| Qr
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: L-def reg-union-def ta-union-gta-lang) fastforce
qed

```

lemma *reg-union-states*:

```

Qr (reg-union A B) = (Inl |∧| Qr A) |∪| (Inr |∧| Qr B)
by (auto simp: reg-union-def)

```

— Deciding emptiness

lemma *ta-empty* [simp]:

```

ta-empty Q A = (gta-lang Q 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 = (L R = {})
by (simp add: L-def reg-empty-def)

```

Epsilon free automaton

lemma *finite-eps-free-rulep* [simp]:

```

finite (Collect (eps-free-rulep A))

```

proof —

```

let ?par = (λ r. case r of f qs → q ⇒ (f, qs)) |∧| (rules A)
let ?st = (λ (r, q). case r of (f, qs) ⇒ TA-rule f qs q) |∧| (?par |×| Q A)
show ?thesis using rule-statesD eps-trancl-statesD
  by (intro finite-subset[of Collect (eps-free-rulep 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 A) (term-of-gterm t) = ta-der 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** $q \in | ?Rs$ **then have** $q \in | ?Ls$
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 $\mathcal{L}\text{-eps-free: } \mathcal{L}\ (eps\ free\ reg\ R) = \mathcal{L}\ R$
by (*auto simp: \mathcal{L}-def eps-free-reg-def*)

Sufficient criterion for containment

definition *ta-contains-aux* :: $(f \times nat)\ set \Rightarrow 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow 'q\ fset \Rightarrow$
bool **where**
 $ta\ contains\ aux\ \mathcal{F}\ Q_1\ \mathcal{A}\ Q_2 \equiv (\forall f\ qs. (f, length\ qs) \in \mathcal{F} \wedge fset\ of\ list\ qs \subseteq | Q_1$
 \longrightarrow
 $(\exists q\ q'. TA\ rule\ f\ qs\ q \in | rules\ \mathcal{A} \wedge q' \in | Q_2 \wedge (q = q' \vee (q, q') \in | (eps$
 $\mathcal{A})|^{+})))$

lemma *ta-contains-aux-state-set:*
assumes $ta\ contains\ aux\ \mathcal{F}\ Q\ \mathcal{A}\ Q\ t \in \mathcal{T}_G\ \mathcal{F}$
shows $\exists q. q \in | Q \wedge q \in | ta\ der\ \mathcal{A}\ (term\ of\ gterm\ t)$ **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** $fset\ of\ list\ qs \subseteq | Q$ $length\ ss = length\ qs$
 $\forall i < length\ qs. qs\ !\ i \in | ta\ der\ \mathcal{A}\ (term\ of\ gterm\ (ss\ !\ i))$
using $ind(4)\ Ex\ list\ of\ length\ P[of\ length\ ss\ \lambda\ q\ i. q \in | Q \wedge q \in | ta\ der\ \mathcal{A}$
 $(term\ of\ gterm\ (ss\ !\ i))]$
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** $Q_2 \subseteq | Q_2'$
shows $ta\ contains\ aux\ \mathcal{F}\ Q_1\ \mathcal{A}\ Q_2 \implies ta\ contains\ aux\ \mathcal{F}\ Q_1\ \mathcal{B}\ Q_2'$
using *assms* **unfolding** *ta-contains-aux-def ta-subset-def*
by (*meson fin-mono francl-mono*)

definition *ta-contains* :: ('f × nat) set ⇒ ('f × nat) set ⇒ ('q, 'f) ta ⇒ 'q fset
 ⇒ 'q fset ⇒ bool

where *ta-contains* $\mathcal{F} \mathcal{G} \mathcal{A} Q Q_f \equiv ta\text{-contains-}aux \mathcal{F} Q \mathcal{A} Q \wedge ta\text{-contains-}aux \mathcal{G} Q \mathcal{A} Q_f$

lemma *ta-contains-mono*:

assumes *ta-subset* $\mathcal{A} \mathcal{B}$ **and** $Q_f \mid\subseteq\mid Q_f'$
shows *ta-contains* $\mathcal{F} \mathcal{G} \mathcal{A} Q Q_f \implies ta\text{-contains} \mathcal{F} \mathcal{G} \mathcal{B} Q Q_f'$
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 $\bigwedge t. \text{groot } t \in \mathcal{G} \implies \bigcup (\text{funas-gterm } 'set (\text{gargs } t)) \subseteq \mathcal{F} \implies t \in \text{gta-lang } Q_f \mathcal{A}$

proof –

fix $t :: 'a \text{ gterm}$
assume $F: \bigcup (\text{funas-gterm } 'set (\text{gargs } t)) \subseteq \mathcal{F}$ **and** $G: \text{groot } t \in \mathcal{G}$
obtain $g \text{ ss}$ **where** $t[simp]: t = GFun \ g \ \text{ss}$ **by** (*cases t, auto*)
then have $\exists \text{qs. length qs} = \text{length ss} \wedge (\forall i < \text{length qs. qs } ! \ i \mid\in\mid \text{ta-der } \mathcal{A} (\text{term-of-gterm } (ss \ ! \ i)) \wedge \text{qs } ! \ i \mid\in\mid Q)$
using *contain ta-contains-aux-state-set*[*of* $\mathcal{F} Q \mathcal{A} \text{ss } ! \ i$ **for** i] F
unfolding *ta-contains-def* $\mathcal{T}_G\text{-funas-gterm-conv}$
using *Ex-list-of-length-P*[*of length ss* $\lambda q \ i. q \mid\in\mid Q \wedge q \mid\in\mid \text{ta-der } \mathcal{A} (\text{term-of-gterm } (ss \ ! \ i))$]
by *auto* (*metis SUP-le-iff nth-mem*)
then obtain qs **where** $\text{length qs} = \text{length ss}$
 $\forall i < \text{length qs. qs } ! \ i \mid\in\mid \text{ta-der } \mathcal{A} (\text{term-of-gterm } (ss \ ! \ i))$
 $\forall i < \text{length qs. qs } ! \ i \mid\in\mid Q$
by *blast*
then obtain q **where** $q \mid\in\mid Q_f \ q \mid\in\mid \text{ta-der } \mathcal{A} (\text{term-of-gterm } t)$
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 $t \in \text{gta-lang } Q_f \mathcal{A}$

by (*intro gta-langI*) *simp*

qed

lemma *ta-contains*:

assumes *contain*: *ta-contains* $\mathcal{F} \mathcal{F} \mathcal{A} Q Q_f$
shows $\mathcal{T}_G \mathcal{F} \subseteq \text{gta-lang } Q_f \mathcal{A}$ (**is** $?A \subseteq -$)

proof –

have [*simp*]: $\text{funas-gterm } t \subseteq \mathcal{F} \implies \text{groot } t \in \mathcal{F}$ **for** t **by** (*cases t*) *auto*
have [*simp*]: $\text{funas-gterm } t \subseteq \mathcal{F} \implies \bigcup (\text{funas-gterm } 'set (\text{gargs } t)) \subseteq \mathcal{F}$ **for** t
by (*cases t*) *auto*
show *thesis* **using** *ta-contains-both*[*OF contain*]
by (*auto simp: 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 $x \in \text{fset-of-list } (\text{sorted-list-of-fset } X)\ y \in \text{fset-of-list } (\text{sorted-list-of-fset } X)$

by *simp-all*

note *this[unfolded mem-idx-fset-sound]*

then have $x = y$ **if** *map-fset-to-nat* $X\ x = \text{map-fset-to-nat } X\ y$

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 $?f = \text{map-fset-fset-to-nat } X$

{ **fix** $x\ y$ **assume** $x \in X\ y \in X$

then have *sorted-list-of-fset* $x \in \text{fset-of-list } (\text{sorted-list-of-fset } (\text{sorted-list-of-fset } |^\dagger X))$

sorted-list-of-fset $y \in \text{fset-of-list } (\text{sorted-list-of-fset } (\text{sorted-list-of-fset } |^\dagger X))$

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 |^\dagger 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* ($Q\ |\cap|\ \mathcal{Q}\ \mathcal{A}$) \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} -relabel [simp]: $\mathcal{L} (\text{relabel-reg } R) = \mathcal{L} R$
by (auto simp: \mathcal{L} -def relabel-reg-def)

lemma relabel-ta-lang [simp]:
 $\text{ta-lang} (\text{relabel-}Q_f Q \mathcal{A}) (\text{relabel-ta } \mathcal{A}) = \text{ta-lang } Q \mathcal{A}$
unfolding ta-lang-to-gta-lang
using relabel-gta-lang
by simp

lemma relabel-fset-gta-lang [simp]:
 $\text{gta-lang} (\text{relabel-fset-}Q_f Q \mathcal{A}) (\text{relabel-fset-ta } \mathcal{A}) = \text{gta-lang } Q \mathcal{A}$
proof –
have $\text{gta-lang} (\text{relabel-fset-}Q_f Q \mathcal{A}) (\text{relabel-fset-ta } \mathcal{A}) = \text{gta-lang} (Q \mid \cap \mid Q \mathcal{A})$
 \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} -reliable-fset [simp]: $\mathcal{L} (\text{reliable-fset-reg } R) = \mathcal{L} R$
by (auto simp: \mathcal{L} -def reliable-fset-reg-def)

lemma ta-states-trim-ta:
 $Q (\text{trim-ta } Q \mathcal{A}) \mid \subseteq \mid Q \mathcal{A}$
unfolding trim-ta-def **using** ta-prod-reach-states .

lemma trim-ta-reach: $Q (\text{trim-ta } Q \mathcal{A}) \mid \subseteq \mid \text{ta-reachable} (\text{trim-ta } Q \mathcal{A})$
unfolding trim-ta-def **using** ta-only-prod-reachable ta-only-reach-reachable
by metis

lemma trim-ta-prod: $Q (\text{trim-ta } Q \mathcal{A}) \mid \subseteq \mid \text{ta-productive } Q (\text{trim-ta } Q \mathcal{A})$
unfolding trim-ta-def **using** ta-only-prod-productive
by metis

lemma empty-gta-lang:
 $\text{gta-lang } Q (TA \{\mid\} \{\mid\}) = \{\}$
using ta-reachable-gtermI
by (force simp: gta-lang-def gta-der-def elim!: ta-langE)

abbreviation empty-reg **where**
 $\text{empty-reg} \equiv \text{Reg } \{\mid\} (TA \{\mid\} \{\mid\})$

lemma \mathcal{L} -empty:
 $\mathcal{L} \text{ empty-reg} = \{\}$
by (auto simp: \mathcal{L} -def empty-gta-lang)

lemma *const-ta-lang*:

gta-lang $\{|q|\}$ (TA $\{| TA\text{-rule } f \ [] \ q \ |\}$ $\{|\|\}$) = $\{GFun\ f \ []\}$

proof –

have [*dest!*]: $q' \ | \in \ | \ ta\text{-der} \ (TA \ \{| \ TA\text{-rule } f \ [] \ q \ |\} \ \{|\|\}) \ t' \implies \ ground \ t' \implies \ t'$
 $= \ Fun \ f \ [] \ \mathbf{for} \ t' \ q'$

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 $\mathcal{A} \ s \ t \implies \ length \ (gargs \ s) = \ length \ (gargs \ t) \wedge (\forall \ i < \ length \ (gargs \ t). \ run \ \mathcal{A} \ (gargs \ s \ ! \ i) \ (gargs \ t \ ! \ i))$

using *run.cases* **by** *fastforce*

lemma *run-root-rule*:

run $\mathcal{A} \ s \ t \implies \ TA\text{-rule} \ (groot\text{-sym} \ t) \ (map \ ex\text{-comp}\text{-state} \ (gargs \ s)) \ (ex\text{-rule}\text{-state} \ s) \ | \in \ | \ (rules \ \mathcal{A}) \wedge$

$(ex\text{-rule}\text{-state} \ s = \ ex\text{-comp}\text{-state} \ s \vee \ (ex\text{-rule}\text{-state} \ s, \ ex\text{-comp}\text{-state} \ s) \ | \in \ | \ (eps \ \mathcal{A})^{|+|})$

by (*cases* s ; *cases* t) (*auto elim: run.cases*)

lemma *run-poss-eq*: *run* $\mathcal{A} \ s \ t \implies \ gposs \ s = \ gposs \ t$

by (*induct rule: run.induct*) *auto*

lemma *run-gsubt-cl*:

assumes *run* $\mathcal{A} \ s \ t$ **and** $p \in \ gposs \ t$

shows *run* $\mathcal{A} \ (gsubt\text{-at} \ s \ p) \ (gsubt\text{-at} \ t \ p)$ **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** $p \in \ gposs \ s$

and $ex\text{-comp}\text{-state} \ (gsubt\text{-at} \ s \ p) = \ ex\text{-comp}\text{-state} \ u$

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 $r \ q \ fs \ ts$ **where** [*simp*]: $s = GFun \ (r, \ q) \ qs \ t = GFun \ fs \ ts$ **by** (*cases* s , *cases* t) *auto*

have $*$: $j < \ length \ qs \implies \ ex\text{-comp}\text{-state} \ (qs[i := (qs \ ! \ i)[p \leftarrow u]_G] \ ! \ j) = \ ex\text{-comp}\text{-state} \ (qs \ ! \ j)$ **for** j

using *Cons(5)* **by** (*cases* $i = j$, *cases* p) *auto*

have [*simp*]: $map \ ex\text{-comp}\text{-state} \ (qs[i := (qs \ ! \ i)[p \leftarrow u]_G]) = \ map \ ex\text{-comp}\text{-state} \ qs$ **using** *Cons(5)*

by (*auto simp: *[unfolded comp-def] intro!: nth-equalityI*)
have $\text{run } \mathcal{A} (qs ! i)[p \leftarrow u]_G (ts ! i)[p \leftarrow v]_G$ **using** *Cons(2-)*
by (*intro Cons(1)*) (*auto dest: run-argsD*)
moreover have $i < \text{length } qs \ i < \text{length } ts$ **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:*

$\text{run } \mathcal{A} s t \implies \text{ex-comp-state } s \mid \in \mid \text{gta-der } \mathcal{A} t$

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 $\text{run } \mathcal{A} s t$ **shows** $\text{ex-rule-state } s \mid \in \mid \text{gta-der } \mathcal{A} t$

proof (*cases s*)

case [*simp*]: (*GFun q qs*)

have $i < \text{length } qs \implies \text{ex-comp-state } (qs ! i) \mid \in \mid \text{gta-der } \mathcal{A} (gargs t ! i)$ **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 $\text{run } \mathcal{A} s t$ **and** $p \in \text{gposs } t$

shows $\text{ex-rule-state } (gsubt\text{-at } s p) \mid \in \mid \text{gta-der } \mathcal{A} (gsubt\text{-at } t p)$

$\text{ex-comp-state } (gsubt\text{-at } s p) \mid \in \mid \text{gta-der } \mathcal{A} (gsubt\text{-at } t p)$

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:*

$q \mid \in \mid \text{gta-der } \mathcal{A} t \implies (\exists p qs. \text{run } \mathcal{A} (GFun (p, q) qs) t)$ **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 \lambda qs i. run \mathcal{A} (GFun (fst qs, ps ! i) (snd qs) (ts ! i))]*

obtain qss **where** $\text{mid: length } qss = \text{length } ts \ \forall i < \text{length } ts. \text{run } \mathcal{A} (GFun (fst (qss ! i), ps ! i) (snd (qss ! i))) (ts ! i)$

by auto

```

have [simp]: map (ex-comp-state  $\circ$  ( $\lambda(qs, y). GFun (fst y, qs) (snd y)$ )) (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 ( $\lambda f args. GFun (fst args, f) (snd args)$ )] ps
qss)
(auto intro: run.intros)
qed

```

```

lemma run-ta-der-ctxt-split1:
assumes run  $\mathcal{A}$  s t p  $\in$  gposs t
shows ex-comp-state s  $\in$  ta-der  $\mathcal{A}$  (ctxt-of-pos-term p (term-of-gterm t)) $\langle$ Var
(ex-comp-state (gsubt-at s p)) $\rangle$ 
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)  $\in$  ta-der  $\mathcal{A}$  (ctxt-of-pos-term p
(term-of-gterm (ts ! i)) $\langle$ Var (ex-comp-state (gsubt-at (qs ! i) p)) $\rangle$ )
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  $\in$  gposs t
shows ex-comp-state s  $\in$  ta-der  $\mathcal{A}$  (ctxt-of-pos-term p (term-of-gterm t)) $\langle$ Var
(ex-rule-state (gsubt-at s p)) $\rangle$ 
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 $\mathcal{A} f ps = (\lambda q'. \exists qs q. TA\text{-rule } f qs q \mid \in \mid \text{rules } \mathcal{A} \wedge \text{list-all2 } (\mid \in \mid) qs ps \wedge (q = q' \vee (q, q') \mid \in \mid (eps \mathcal{A})^+))$

lemma *ps-reachable-statespE*:

assumes *ps-reachable-statesp* $\mathcal{A} f qs q$
obtains *ps p* **where** *TA-rule* $f ps p \mid \in \mid \text{rules } \mathcal{A} \text{list-all2 } (\mid \in \mid) ps qs (p = q \vee (p, q) \mid \in \mid (eps \mathcal{A})^+)$
using *assms* **unfolding** *ps-reachable-statesp-def*
by *auto*

lemma *ps-reachable-statesp-Q*:

ps-reachable-statesp $\mathcal{A} f ps q \implies q \mid \in \mid \mathcal{Q} \mathcal{A}$
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 (Q A)]*)
(*auto simp: ps-reachable-statesp-Q*)

lemmas *finite-Collect-ps-statep-unfolded* [*simp*] = *finite-Collect-ps-statep*[*unfolded ps-reachable-statesp-def, simplified*]

definition *ps-reachable-states* $\mathcal{A} f ps \equiv fCollect (ps-reachable-statesp \mathcal{A} f ps)$

lemmas *ps-reachable-states-simp* = *ps-reachable-statesp-def ps-reachable-states-def*

lemma *ps-reachable-states-fmember*:

$q' \mid \in \mid ps\text{-reachable-states } \mathcal{A} f ps \longleftrightarrow (\exists qs q. TA\text{-rule } f qs q \mid \in \mid \text{rules } \mathcal{A} \wedge \text{list-all2 } (\mid \in \mid) qs ps \wedge (q = q' \vee (q, q') \mid \in \mid (eps \mathcal{A})^+))$
by (*auto simp: ps-reachable-states-simp*)

lemma *ps-reachable-statesI*:

assumes *TA-rule* $f ps p \mid \in \mid \text{rules } \mathcal{A} \text{list-all2 } (\mid \in \mid) ps qs (p = q \vee (p, q) \mid \in \mid (eps \mathcal{A})^+)$
shows $p \mid \in \mid ps\text{-reachable-states } \mathcal{A} f qs$
using *assms* **unfolding** *ps-reachable-states-simp*
by *auto*

lemma *ps-reachable-states-sig*:

ps-reachable-states $\mathcal{A} f ps \neq \{\mid\} \implies (f, \text{length } ps) \mid \in \mid \text{ta-sig } \mathcal{A}$
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 \Rightarrow 'a FSet-Lex-Wrapper fset \Rightarrow 'a FSet-Lex-Wrapper fset \Rightarrow bool

where *ps-states-complete-it* \mathcal{A} Q Q_{next} \equiv
 $\forall f$ ps. fset-of-list ps \subseteq $Q \wedge$ ps-reachable-states \mathcal{A} f (map ex ps) \neq $\{\|\}$ \longrightarrow
 Wrapp (ps-reachable-states \mathcal{A} f (map ex ps)) \subseteq Q_{next}

lemma *ps-states-complete-itD*:

ps-states-complete-it \mathcal{A} Q Q_{next} \implies fset-of-list ps \subseteq $Q \implies$
 ps-reachable-states \mathcal{A} f (map ex ps) \neq $\{\|\}$ \implies Wrapp (ps-reachable-states \mathcal{A}
 f (map ex ps)) \subseteq Q_{next}

unfolding *ps-states-complete-it-def* **by** *blast*

abbreviation *ps-states-complete* \mathcal{A} $Q \equiv$ *ps-states-complete-it* \mathcal{A} Q Q

The least complete set of states

inductive-set *ps-states-set* **for** \mathcal{A} **where**

$\forall p \in$ set ps. $p \in$ *ps-states-set* $\mathcal{A} \implies$ ps-reachable-states \mathcal{A} f (map ex ps) \neq $\{\|\}$
 \implies
 Wrapp (ps-reachable-states \mathcal{A} f (map ex ps)) \in *ps-states-set* \mathcal{A}

lemma *ps-states-Pow*:

ps-states-set $\mathcal{A} \subseteq$ fset (Wrapp \mid fPow (\mathcal{Q} \mathcal{A}))

proof –

{**fix** q **assume** $q \in$ *ps-states-set* \mathcal{A} **then have** $q \in$ fset (Wrapp \mid fPow (\mathcal{Q} \mathcal{A}))
by *induct* (auto simp: ps-reachable-states-p- \mathcal{Q} ps-reachable-states-def im-
 age-iff)}

then show ?thesis **by** *blast*

qed

context

includes *fset.lifting*

begin

lift-definition *ps-states* :: ('a, 'b) ta \Rightarrow 'a FSet-Lex-Wrapper fset **is** *ps-states-set*
by (auto intro: finite-subset[OF *ps-states-Pow*])

lemma *ps-states*: *ps-states* $\mathcal{A} \subseteq$ Wrapp \mid 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 Q_{next} $Q_{next} \subseteq Q$
shows *ps-states* $\mathcal{A} \subseteq Q$

proof *standard*

fix q **assume** *ass*: $q \in \text{ps-states } \mathcal{A}$ **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* \Rightarrow 'a *FSet-Lex-Wrapper* *fset* \Rightarrow ('a *FSet-Lex-Wrapper*,
'b) *ta-rule* \Rightarrow *bool* **where**

ps-rulesp \mathcal{A} $Q = (\lambda r. \exists f ps p. r = \text{TA-rule } f ps (\text{Wrapp } p) \wedge \text{fset-of-list } ps \subseteq Q \wedge$
 $p = \text{ps-reachable-states } \mathcal{A} f (\text{map } ex ps) \wedge p \neq \{\}\})$

definition *ps-rules* **where**

ps-rules \mathcal{A} $Q \equiv \text{fCollect } (\text{ps-rulesp } \mathcal{A} Q)$

lemma *finite-ps-rulesp* [*simp*]:

finite (*Collect* (*ps-rulesp* \mathcal{A} Q)) (**is** *finite* ?*S*)

proof –

let ?*Q* = *fset* (*Wrapp* | \uparrow *fPow* (Q \mathcal{A}) | \cup | Q) **let** ?*sig* = *fset* (*ta-sig* \mathcal{A})
define *args* **where** *args* $\equiv \bigcup (f, n) \in ?\text{sig}. \{qs \mid \text{set } qs \subseteq ?Q \wedge \text{length } qs = n\}$

define *bound* **where** *bound* $\equiv \bigcup (f, -) \in ?\text{sig}. \bigcup q \in ?Q. \bigcup qs \in \text{args}. \{\text{TA-rule } f qs q\}$

have *finite*: *finite* ?*Q* *finite* ?*sig* **by** (*auto intro: finite-subset*)

then have *finite args* **using** *finite-lists-length-eq*[*OF* $\langle \text{finite } ?Q \rangle$]

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) \subseteq *bound*

proof *standard*

fix r **assume** *: $r \in \text{Collect } (\text{ps-rulesp } \mathcal{A} Q)$

obtain $f ps p$ **where** $r[\text{simp}]$: $r = \text{TA-rule } f ps p$ **by** (*cases* r)

from * **obtain** $qs q$ **where** $\text{TA-rule } f qs q \in \text{rules } \mathcal{A}$ **and** *len*: $\text{length } ps = \text{length } qs$

unfolding *ps-rulesp-def* *ps-reachable-states-simp*

using *list-all2-lengthD* **by** *fastforce*

from this have *sym*: $(f, \text{length } qs) \in ?\text{sig}$

by *auto*

moreover from * **have** $\text{set } ps \subseteq ?Q$ **unfolding** *ps-rulesp-def*

by (*auto simp flip: fset-of-list-elem simp: ps-reachable-statesp-def*)

ultimately have $ps: ps \in \text{args}$

by (*auto simp only: args-def UN-iff intro!: beXI*[*of - (f, length qs)*] *len*)

from * **have** $p \in ?Q$ **unfolding** $ps\text{-rulesp-def}$ $ps\text{-reachable-states-def}$
using $ps\text{-reachable-statesp-Q}$
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 $finite\text{-ps-rulesp-unfolded} = finite\text{-ps-rulesp}[unfolding\ ps\text{-rulesp-def},\ simplified]$

lemmas $ps\text{-rulesI}$ [*intro!*] = $fCollect\text{-memberI}[OF\ finite\text{-ps-rulesp}]$

lemma $ps\text{-rules-states}$:

$rule\text{-states}(fCollect\ (ps\text{-rulesp}\ \mathcal{A}\ Q)) \mid\subseteq\ (Wrapp\ |\cdot|\ fPow\ (Q\ \mathcal{A})\ |\cup|\ Q)$
by (*auto simp: ps-rulesp-def rule-states-def ps-reachable-states-def ps-reachable-statesp-Q*)
blast

definition $ps\text{-ta} :: ('q, 'f)\ ta \Rightarrow ('q\ FSet\text{-Lex}\text{-Wrapper}, 'f)\ ta$ **where**

$ps\text{-ta}\ \mathcal{A} = (let\ Q = ps\text{-states}\ \mathcal{A}\ in$
 $TA\ (ps\text{-rules}\ \mathcal{A}\ Q)\ \{\{\}\})$

definition $ps\text{-ta}\text{-}Q_f :: 'q\ fset \Rightarrow ('q, 'f)\ ta \Rightarrow 'q\ FSet\text{-Lex}\text{-Wrapper}\ fset$ **where**

$ps\text{-ta}\text{-}Q_f\ Q\ \mathcal{A} = (let\ Q' = ps\text{-states}\ \mathcal{A}\ in$
 $ffilter\ (\lambda\ S.\ Q\ \mid\cap\ (ex\ S) \neq \{\{\}\})\ Q')$

lemma $ps\text{-rules-sound}$:

assumes $p \mid\in\ ta\text{-der}\ (ps\text{-ta}\ \mathcal{A})$ (*term-of-gterm t*)
shows $ex\ p \mid\subseteq\ ta\text{-der}\ \mathcal{A}$ (*term-of-gterm t*) **using** *assms*
proof (*induction rule: ta-der-gterm-induct*)
case ($GFun\ f\ ts\ ps\ p\ q$)
then have $IH: \forall i < length\ ts.\ ex\ (ps\ !\ i) \mid\subseteq\ gta\text{-der}\ \mathcal{A}\ (ts\ !\ i)$ **unfolding**
 $gta\text{-der-def}$ **by** *auto*
show *?case*
proof *standard*
fix r **assume** $r \mid\in\ ex\ q$
with $GFun(1 - 3)$ **obtain** $qs\ q'$ **where** $TA\text{-rule}\ f\ qs\ q' \mid\in\ rules\ \mathcal{A}$
 $q' = r \vee (q', r) \mid\in\ (eps\ \mathcal{A})^+ \mid list\text{-all2}\ (\mid\in\)\ qs\ (map\ ex\ ps)$
by (*auto simp: Let-def ps-ta-def ps-rulesp-def ps-reachable-states-simp ps-rules-def*)
then show $r \mid\in\ ta\text{-der}\ \mathcal{A}$ (*term-of-gterm (GFun f ts)*)
using $GFun(2)$ IH **unfolding** $gta\text{-der-def}$
by (*force dest!: fsubsetD intro!: exI[of - q'] exI[of - qs] simp: list-all2-conv-all-nth*)
qed
qed

lemma $ps\text{-ta-nt-empty-set}$:

$TA\text{-rule}\ f\ qs\ (Wrapp\ \{\{\}\}) \mid\in\ rules\ (ps\text{-ta}\ \mathcal{A}) \Longrightarrow False$

by (auto simp: ps-ta-def ps-rulesp-def ps-rules-def)

lemma ps-rules-not-empty-reach:

assumes Wrapp $\{\|\}$ $|\in|$ ta-der (ps-ta \mathcal{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 - \mathcal{A}]

by (auto simp: ps-ta-def)

qed

lemma ps-rules-complete:

assumes q $|\in|$ ta-der \mathcal{A} (term-of-gterm t)

shows $\exists p. q$ $|\in|$ ex $p \wedge p$ $|\in|$ ta-der (ps-ta \mathcal{A}) (term-of-gterm t) $\wedge p$ $|\in|$ ps-states \mathcal{A} using assms

proof (induction rule: ta-der-gterm-induct)

let $?P = \lambda t q p. q$ $|\in|$ ex $p \wedge p$ $|\in|$ ta-der (ps-ta \mathcal{A}) (term-of-gterm t) $\wedge p$ $|\in|$ ps-states \mathcal{A}

case (GFun f ts ps p q)

then have $\forall i. \exists p. i < \text{length } ts \longrightarrow ?P (ts ! i) (ps ! i) p$ by auto

with choice[OF this] obtain psf where $ps: \forall i < \text{length } ts. ?P (ts ! i) (ps ! i) (psf i)$ by auto

define qs where $qs = \text{map } psf [0 .. < \text{length } ts]$

let $?p = \text{ps-reachable-states } \mathcal{A} f (\text{map } ex qs)$

from ps have in-Q: fset-of-list qs $|\subseteq|$ ps-states \mathcal{A}

by (auto simp: qs-def fset-of-list-elem)

from ps GFun(2) have all: list-all2 ($|\in|$) ps (map ex qs)

by (auto simp: list-all2-conv-all-nth qs-def)

then have in-p: q $|\in|$? 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) $|\in|$ ps-rules \mathcal{A} (ps-states \mathcal{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 $|\in|$ (ps-states \mathcal{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 \mathcal{A}) = $\{\|\}$ by (auto simp: Let-def ps-ta-def)

lemma ps-ta-det[iff]: ta-det (ps-ta \mathcal{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 \subseteq ? R **proof** standard

fix t assume $t \in ?L$

then obtain q **where** $q\text{-res}: q \in | \text{ta-der } \mathcal{A} \text{ (term-of-gterm } t) \text{ and } q\text{-final}: q \in | Q$
by *auto*
from $ps\text{-rules-complete}[OF\ q\text{-res}]$ **obtain** p **where**
 $p \in | ps\text{-states } \mathcal{A} \ q \in | \text{ex } p \ p \in | \text{ta-der } (ps\text{-ta } \mathcal{A}) \text{ (term-of-gterm } t)$
by *auto*
moreover with $q\text{-final}$ **have** $p \in | ps\text{-ta-}Q_f \ Q \ \mathcal{A}$
by $(\text{auto simp: } ps\text{-ta-}Q_f\text{-def})$
ultimately show $t \in ?R$ **by** *auto*
qed
show $?R \subseteq ?L$ **proof** *standard*
fix t **assume** $t \in ?R$
then obtain p **where**
 $p\text{-res}: p \in | \text{ta-der } (ps\text{-ta } \mathcal{A}) \text{ (term-of-gterm } t) \text{ and } p\text{-final}: p \in | ps\text{-ta-}Q_f \ Q$
 \mathcal{A}
by $(\text{auto simp: } \text{ta-lang-def})$
from $ps\text{-rules-sound}[OF\ p\text{-res}]$ **have** $\text{ex } p \ | \subseteq | \text{ta-der } \mathcal{A} \text{ (term-of-gterm } t)$
by *auto*
moreover from $p\text{-final}$ **obtain** q **where** $q \in | \text{ex } p \ q \in | Q$ **by** $(\text{auto simp: } ps\text{-ta-}Q_f\text{-def})$
ultimately show $t \in ?L$ **by** *auto*
qed
qed

definition $ps\text{-reg}$ **where**
 $ps\text{-reg } R = \text{Reg } (ps\text{-ta-}Q_f \ (\text{fin } R) \ (\text{ta } R)) \ (ps\text{-ta } (\text{ta } R))$

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

lemma $ps\text{-ta-states}$: $Q \ (ps\text{-ta } \mathcal{A}) \ | \subseteq | \text{Wrapp } | \uparrow \ fPow \ (Q \ \mathcal{A})$
using $ps\text{-rules-states } ps\text{-states}$ **unfolding** $ps\text{-ta-def } Q\text{-def}$
by $(\text{auto simp: } \text{Let-def } ps\text{-rules-def})$ *force*

lemma $ps\text{-ta-states}'$: $\text{ex } | \uparrow \ Q \ (ps\text{-ta } \mathcal{A}) \ | \subseteq | \ fPow \ (Q \ \mathcal{A})$
using $ps\text{-ta-states}[of \ \mathcal{A}]$
by *fastforce*

end
theory *Tree-Automata-Complement*
imports *Tree-Automata-Det*
begin

3.3 Complement closure of regular languages

definition $partially\text{-completely-defined-on}$ **where**
 $partially\text{-completely-defined-on } \mathcal{A} \ \mathcal{F} \longleftrightarrow$
 $(\forall t. \text{fmas-gterm } t \subseteq \text{fset } \mathcal{F} \longleftrightarrow (\exists q. q \in | \text{ta-der } \mathcal{A} \text{ (term-of-gterm } t)))$

definition *sig-ta where*

sig-ta $\mathcal{F} = TA ((\lambda (f, n). TA\text{-rule } f (\text{replicate } n ()) ()) \mid \uparrow \mathcal{F}) \{\mid\}$

lemma *sig-ta-rules-fmember:*

TA-rule $f \text{ } qs \text{ } q \mid \in \mid \text{ rules } (sig\text{-ta } \mathcal{F}) \longleftrightarrow (\exists n. (f, n) \mid \in \mid \mathcal{F} \wedge qs = \text{replicate } n () \wedge q = ())$

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* $t \subseteq \text{fset } \mathcal{F}$
then have $() \mid \in \mid \text{ ta-der } (sig\text{-ta } \mathcal{F}) (\text{term-of-gterm } t)$

proof (*induct* t)

case (*GFun* $f \text{ } ts$)

then show *?case*

by (*auto simp: sig-ta-rules-fmember SUP-le-iff*
intro!: exI[of - replicate (length ts) ()])

qed}

moreover

{fix $t \text{ } q$ **assume** $q \mid \in \mid \text{ ta-der } (sig\text{-ta } \mathcal{F}) (\text{term-of-gterm } t)$

then have *funas-gterm* $t \subseteq \text{fset } \mathcal{F}$

proof (*induct rule: ta-der-gterm-induct*)

case (*GFun* $f \text{ } ts \text{ } ps \text{ } p \text{ } q$)

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* $\mathcal{A} \mid \subseteq \mid \mathcal{F}$

shows *partially-completely-defined-on* \mathcal{A} \mathcal{F}

proof –

have *ta-der* (*sig-ta* \mathcal{F}) $t \mid \subseteq \mid \text{ ta-der } \mathcal{A} \text{ } t$ **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} $\mathcal{F} \implies \text{partially-completely-defined-on } (ps\text{-ta } \mathcal{A}) \mathcal{F}$

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-defined-ta-unionI*:

partially-completely-defined-on A F \implies *ta-sig B* $|\subseteq|$ *F* \implies $\mathcal{Q} \mathcal{A} \mid \cap \mid \mathcal{Q} \mathcal{B} = \{\mid\}$
 \implies

partially-completely-defined-on (ta-union A B) F
unfolding *partially-completely-defined-on-def*
using *ta-union-der-disj-states'*[*of A 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-defined-fmaps-statesI*:

partially-completely-defined-on A F \implies *finj-on f* ($\mathcal{Q} \mathcal{A}$) \implies *partially-completely-defined-on*
(*fmap-states-ta f A*) *F*

unfolding *partially-completely-defined-on-def*
using *fsubsetD*[*OF ta-der-fmap-states-ta-mono2, of f 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 A F ta-det A*
shows *gta-lang* ($\mathcal{Q} \mathcal{A} \mid - \mid \mathcal{Q}$) $\mathcal{A} = \mathcal{T}_G$ (*fset F*) $-$ *gta-lang* $\mathcal{Q} \mathcal{A}$ (**is** $?Ls = ?Rs$)

proof $-$

{**fix** *t* **assume** $t \in ?Ls$
then obtain *p* **where** $p: p \mid \in \mid \mathcal{Q} \mathcal{A} \ p \mid \notin \mid \mathcal{Q} \ p \mid \in \mid ta-der \ \mathcal{A}$ (*term-of-gterm t*)
by *auto*
from *ta-detE*[*OF assms(2) p(3)*] **have** $\forall q. q \mid \in \mid ta-der \ \mathcal{A}$ (*term-of-gterm t*)
 $\longrightarrow q = p$
by *blast*
moreover have *funas-gterm t* \subseteq *fset 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 $t \in ?Rs$ **using** $p(2)$
by (*auto simp: T_G-equivalent-def*)}

moreover
{**fix** *t* **assume** $t \in ?Rs$
then have $f: funas-gterm \ t \subseteq fset \ F \ \forall q. q \mid \in \mid ta-der \ \mathcal{A}$ (*term-of-gterm t*) \longrightarrow
 $q \mid \notin \mid \mathcal{Q}$
by (*auto simp: T_G-equivalent-def*)
from $f(1)$ **obtain** *p* **where** $p \mid \in \mid ta-der \ \mathcal{A}$ (*term-of-gterm t*) **using** *assms(1)*
by (*force simp: partially-completely-defined-on-def*)
then have $t \in ?Ls$ **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*:

$q \mid \in \mid ta-der \ \mathcal{A}$ (*term-of-gterm t*) \implies *funas-gterm t* \subseteq *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 $\mathcal{F} \mathcal{A} = TA$ (*ffilter* ($\lambda r. (r\text{-root } r, \text{length } (r\text{-lhs-states } r)) \mid \in \mid \mathcal{F}$) (*rules* \mathcal{A})) (*eps* \mathcal{A})

definition *filter-ta-reg* **where**

filter-ta-reg $\mathcal{F} R = Reg$ (*fin* R) (*filter-ta-sig* \mathcal{F} (*ta* R))

lemma *filter-ta-sig*:

ta-sig (*filter-ta-sig* $\mathcal{F} \mathcal{A}$) $\mid \subseteq \mid \mathcal{F}$
by (*auto simp: ta-sig-def filter-ta-sig-def*)

lemma *filter-ta-sig-lang*:

gta-lang Q (*filter-ta-sig* $\mathcal{F} \mathcal{A}$) = *gta-lang* $Q \mathcal{A} \cap \mathcal{T}_G$ (*fset* \mathcal{F}) (**is** $?Ls = ?Rs$)

proof –

let $?A = \text{filter-ta-sig } \mathcal{F} \mathcal{A}$
{fix t **assume** $t \in ?Ls$
then obtain q **where** $q: q \mid \in \mid Q \ q \mid \in \mid \text{ta-der } ?A$ (*term-of-gterm* t)
by *auto*
then have *funas-gterm* $t \subseteq \text{fset } \mathcal{F}$
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 $t \in ?Rs$ **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
 $\mathcal{A} = ?A]$)}
moreover
{fix t **assume** *ass: t* $t \in ?Rs$
then have *funas: funas-gterm* $t \subseteq \text{fset } \mathcal{F}$
by (*auto simp: T_G-equivalent-def*)
from *ass* **obtain** p **where** $p: p \mid \in \mid Q \ p \mid \in \mid \text{ta-der } \mathcal{A}$ (*term-of-gterm* t)
by *auto*
from *this(2)* *funas* **have** $p \mid \in \mid \text{ta-der } ?A$ (*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 $t \in ?Ls$ **using** $p(1)$ **by** *auto*}
ultimately show $?thesis$ **by** *blast*
qed

lemma \mathcal{L} -*filter-ta-reg*:

\mathcal{L} (*filter-ta-reg* $\mathcal{F} \mathcal{A}$) = $\mathcal{L} \mathcal{A} \cap \mathcal{T}_G$ (*fset* \mathcal{F})
using *filter-ta-sig-lang*
by (*auto simp: L-def filter-ta-reg-def*)

definition *sig-ta-reg* **where**

sig-ta-reg $\mathcal{F} = \text{Reg } \{\{\}\}$ (*sig-ta* \mathcal{F})

lemma *L-sig-ta-reg*:

\mathcal{L} (*sig-ta-reg* \mathcal{F}) = $\{\}$

by (*auto simp: L-def sig-ta-reg-def*)

definition *complement-reg* **where**

complement-reg $R \mathcal{F} = (\text{let } \mathcal{A} = \text{ps-reg } (\text{reg-union } (\text{sig-ta-reg } \mathcal{F}) R) \text{ in } \text{Reg } (\mathcal{Q}_r \mathcal{A} \mid - \mid \text{fin } \mathcal{A}) (\text{ta } \mathcal{A}))$

lemma *L-complement-reg*:

assumes *ta-sig* (*ta* \mathcal{A}) $\mid \subseteq \mid \mathcal{F}$

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 *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: L-ps-reg L-union L-sig-ta-reg*)

qed

lemma *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* $\mathcal{F} \mathcal{A}$)) $\mid \subseteq \mid \mathcal{F}$

by (*auto simp: filter-ta-reg-def filter-ta-sig*)

show *?thesis* **unfolding** *L-complement-reg*[*OF* $*$] *L-filter-ta-reg*

by *blast*

qed

definition *difference-reg* **where**

difference-reg $R L = (\text{let } F = \text{ta-sig } (\text{ta } R) \text{ in } \text{reg-intersect } R (\text{trim-reg } (\text{complement-reg } (\text{filter-ta-reg } F L) F)))$

lemma *L-difference-reg*:

\mathcal{L} (*difference-reg* $R L$) = $\mathcal{L} R$ - $\mathcal{L} L$ (**is** *?Ls* = *?Rs*)

unfolding *difference-reg-def Let-def L-trim L-intersect 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* $ts\ Cs \equiv \text{Suc} (\text{length } Cs) = \text{length } ts \wedge$
 $(\forall i < \text{length } Cs. (Cs ! i) \langle ts ! i \rangle = ts ! \text{Suc } i)$

abbreviation *derivation-ctxt-st* $A\ ts\ Cs\ qs \equiv \text{length } qs = \text{length } ts \wedge \text{Suc} (\text{length } Cs) = \text{length } ts \wedge$
 $(\forall i < \text{length } Cs. qs ! \text{Suc } i \in | \text{ta-der } A (Cs ! i) \langle \text{Var } (qs ! i) \rangle)$

abbreviation *derivation-sound* $A\ ts\ qs \equiv \text{length } qs = \text{length } ts \wedge$
 $(\forall i < \text{length } qs. qs ! i \in | \text{ta-der } A (ts ! i))$

definition *derivation* $A\ ts\ Cs\ qs \longleftrightarrow \text{derivation-ctxt } ts\ Cs \wedge$
 $\text{derivation-ctxt-st } A\ ts\ Cs\ qs \wedge \text{derivation-sound } A\ ts\ qs$

lemma *ctxt-comp-lhs-not-hole*:

assumes $C \neq \square$
shows $C \circ_c D \neq \square$
using *assms* **by** (*cases C*; *cases D*) *auto*

lemma *ctxt-comp-rhs-not-hole*:

assumes $D \neq \square$
shows $C \circ_c D \neq \square$
using *assms* **by** (*cases C*; *cases D*) *auto*

lemma *fold-ctxt-comp-nt-empty-acc*:

assumes $D \neq \square$
shows $\text{fold } (\circ_c) Cs\ D \neq \square$
using *assms* **by** (*induct Cs arbitrary: D*) (*auto simp add: ctxt-comp-rhs-not-hole*)

lemma *fold-ctxt-comp-nt-empty*:

assumes $C \in \text{set } Cs$ **and** $C \neq \square$
shows $\text{fold } (\circ_c) Cs\ D \neq \square$ **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]*:

$\square \wedge^n = \square$
by (*induct n*) *auto*

lemma *ctxt-comp-not-hole*:

assumes $C \neq \square$ **and** $n \neq 0$
shows $C \wedge^n \neq \square$
using *assms* **by** (*induct n arbitrary: C*) (*auto simp: ctxt-comp-lhs-not-hole*)

lemma *ctxt-comp-n-suc [simp]*:

shows $(C \wedge (\text{Suc } n)) \langle t \rangle = (C \wedge n) \langle C \langle t \rangle \rangle$

by (induct n arbitrary: C) auto

lemma *ctxt-comp-reach*:

assumes $p \in | \text{ta-der } A \ C \langle \text{Var } p \rangle$

shows $p \in | \text{ta-der } A \ (C \hat{=} n) \langle \text{Var } p \rangle$

using *assms* by (induct n arbitrary: C) (auto intro: ta-der-ctxt)

lemma *args-depth-less* [*simp*]:

assumes $u \in \text{set } ss$

shows $\text{depth } u < \text{depth } (\text{Fun } f \ ss)$ using *assms*

by (cases ss) (auto simp: less-Suc-eq-le)

lemma *subterm-depth-less*:

assumes $s \triangleright t$

shows $\text{depth } t < \text{depth } s$

using *assms* by (induct s t rule: supt.induct) (auto intro: less-trans)

lemma *poss-length-depth*:

shows $\exists p \in \text{poss } t. \text{length } p = \text{depth } t$

proof (induct t)

case (Fun f ts)

then show ?case

proof (cases ts)

case [*simp*]: (Cons a list)

have $ts \neq [] \implies \exists s. f \ s = \text{Max } (f \ ' \ \text{set } ts) \wedge s \in \text{set } ts$ **for** $ts \ f$

using *Max-in*[of $f \ ' \ \text{set } ts$] **by** (auto simp: image-iff)

from *this*[of $ts \ \text{depth}$] **obtain** s **where** $s: \text{depth } s = \text{Max } (\text{depth } \ ' \ \text{set } ts) \wedge s \in$

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 $p \in \text{poss } t$

shows $\text{length } p \leq \text{depth } t$ 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 $C \neq []$

shows $\text{depth } t < \text{depth } C \langle t \rangle$ 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⟨t⟩ using depth-ctxt-nt-hole-inc less-imp-le
by (*cases C, simp*) *blast*

lemma *ctxt-comp-n-not-hole-depth-inc*:
assumes C ≠ □
shows depth (C[^]n)⟨t⟩ < depth (C[^](Suc n))⟨t⟩
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 ≠ □
shows n < depth (C[^](Suc n))⟨t⟩
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 A t q |∈| ta-der A C⟨Var q⟩
shows q |∈| ta-der A (C[^]n)⟨t⟩
using assms by (induct n) (auto simp: ta-der-ctxt)

lemma *ctxt-compose-funs-ctxt [simp]*:
funs-ctxt (C ◦_c D) = funs-ctxt C ∪ funs-ctxt D
by (*induct C arbitrary: D*) *auto*

lemma *ctxt-compose-vars-ctxt [simp]*:
vars-ctxt (C ◦_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[^]n) = {} vars-ctxt (C[^]n) = {}
using assms by auto

lemma *ctxt-power-funs-vars-n [simp]*:
assumes n ≠ 0
shows funs-ctxt (C[^]n) = funs-ctxt C vars-ctxt (C[^]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 ∈ poss s
  shows p ∈ 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 ∈ poss t and s ∈ set (terms-pos t p)
  shows t ⊇ s using assms
  using assms
proof (induct p arbitrary: t s)
  case (Cons a p)
  from Cons(2) have st: t ⊇ t |- [a]
  by (simp add: subt-at-imp-supteq)
  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 ∈ 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 < terms-pos t p ! j
  using assms
proof (induct p arbitrary: t i j)
  case (Cons a p)
  from Cons(2-) have t |- [a] ⊇ 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 ap-
  pend-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  $\bigwedge i. i < \text{Suc } (\text{length } p) \implies t \triangleright (\text{terms-pos } (t \text{ |- } [a]) p) ! i$ 
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  $\exists p \in \text{poss } t. \text{length } (\text{terms-pos } t p) = (n + 1)$ 
proof -
obtain p where p: p  $\in \text{poss } t$  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  $\in \text{poss } t$  and q  $|\in| \text{ta-der } A t$ 
shows  $\exists Cs \text{ } qs. \text{derivation } A (\text{terms-pos } t p) Cs \text{ } qs \wedge \text{last } qs = q$ 
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  $\in \text{poss } (t \text{ |- } [a])$  by auto
from Cons(2) obtain C where C: C (t |- [a]) = t
by (meson ctxt-supt-id empty-pos-in-poss poss-Cons-poss)
from C ta-der-ctxt-decompose Cons(3) obtain q' where
res: q'  $|\in| \text{ta-der } A (t \text{ |- } [a])$  q  $|\in| \text{ta-der } A C \langle \text{Var } q' \rangle$ 
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  $\wedge \text{last } qs = q'$ 
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.premis(2) nth-append-Cons)

qed

lemma *derivation-ctxt-terms-pos-nt-empty:*

assumes $p \in \text{poss } t$ and *derivation-ctxt* (terms-pos t p) Cs and $C \in \text{set } Cs$
 shows $C \neq \square$

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 $p \in \text{poss } t$ and *derivation-ctxt* (terms-pos t p) Cs

and $i < \text{length } Cs$ and $j \leq \text{length } Cs$ and $i < j$

shows $\text{fold } (\circ_c) (\text{take } (j - i) (\text{drop } i Cs)) \square \neq \square$

proof –

have $\exists C. C \in \text{set } (\text{take } (j - i) (\text{drop } i Cs))$

using *assms*(3–) not-le by fastforce

then obtain C where $w: C \in \text{set } (\text{take } (j - i) (\text{drop } i Cs))$ by blast

then have $C \neq \square$

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 $i < \text{length } Cs$ and $j \leq \text{length } Cs$ and $i < j$

shows $(\text{fold } (\circ_c) (\text{take } (j - i) (\text{drop } i Cs)) \square) \langle ts ! i \rangle = ts ! j$

using *assms*

proof (induct $j - i$ arbitrary: j i)

case (Suc x)

then obtain n where j [simp]: $j = \text{Suc } n$ 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 $i < \text{length } Cs$ and $j \leq \text{length } Cs$ and $i < j$

shows $qs ! j \in | \text{ta-der } A (\text{fold } (\circ_c) (\text{take } (j - i) (\text{drop } i Cs)) \square) \langle \text{Var } (qs ! i) \rangle$

using *assms*

proof (induct $j - i$ arbitrary: j i)

case (Suc x)

then obtain n where j [simp]: $j = \text{Suc } n$ 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 ∈ poss t*
shows $\forall s \in \text{set } (terms\text{-pos } t p). \text{ground } s$
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 $\exists i < \text{length } qs. \exists j < \text{length } qs. i < j \wedge qs ! i = qs ! j$
using *assms* **by** *auto (metis distinct-card distinct-conv-nth linorder-neqE-nat)*

lemma *length-remdups-less-eq*:
assumes *set xs ⊆ set ys*
shows $\text{length } (\text{remdups } xs) \leq \text{length } (\text{remdups } ys)$ **using** *assms*
by (*auto simp: length-remdups-card-conv card-mono*)

lemma *pigeonhole-tree-automata*:
assumes *fcard (Q A) < depth t and q |∈| ta-der A t and ground t*
shows $\exists C \ C2 \ v \ p. C2 \neq \square \wedge C \langle C2 \langle v \rangle \rangle = t \wedge p \ | \in | \text{ta-der } A \ v \wedge$
 $p \ | \in | \text{ta-der } A \ C2 \langle \text{Var } p \rangle \wedge q \ | \in | \text{ta-der } A \ C \langle \text{Var } p \rangle$
proof –
obtain *p n* **where** *p: p ∈ poss t depth t = n and*
 $\text{card: fcard } (Q \ A) < n \text{ length } (terms\text{-pos } t \ p) = (n + 1)$
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
 $\text{derivation: derivation } A \ (terms\text{-pos } t \ p) \ Cs \ qs \wedge \text{last } qs = q$ **by** *blast*
then have *d-ctxt: derivation-ctxt-st A (terms-pos t p) Cs qs derivation-ctxt*
 $(terms\text{-pos } t \ p) \ Cs$
by (*auto simp: derivation-def*)
then have *l: length Cs = length qs - 1* **by** (*auto simp: derivation-def*)
from *derivation* **have** *sub: fset-of-list qs |⊆| Q A length qs = length (terms-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*
 $\text{in-fset-conv-nth nth-mem})$
then have $\exists i < \text{length } (\text{butlast } qs). \exists j < \text{length } (\text{butlast } qs). i < j \wedge (\text{butlast}$
 $qs) ! i = (\text{butlast } qs) ! j$
using *card(1, 2) assms(1) fcard-mono[OF sub(1)] length-remdups-less-eq[of*
 $\text{butlast } qs \ qs]$

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 (o_c) (take (j - i) (drop i Cs)) □ ≠ □*
using *derivation-ctxt-terms-pos-sub-list-nt-empty*[*OF p(1) d-ctxt(2) len(1) or-*
der.strict-implies-order[*OF len(2)*] *less*] .
moreover have (*fold (o_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 (o_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 (o_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 (o_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 A (term-of-gterm s) = ta-der A (term-of-gterm t)*
shows *ta-der A (term-of-gterm C⟨s⟩_G) = ta-der A (term-of-gterm C⟨t⟩_G)* **using**
assms
by (*induct C*) (*auto, (metis append-Cons-nth-not-middle nth-append-length)+*)

locale *myhill-nerode =*

fixes *F L* **assumes** *term-subset: L ⊆ T_G F*
begin

definition *myhill* ($\langle \cdot \equiv_{\mathcal{L}} \cdot \rangle$) **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*: $s \equiv_{\mathcal{L}} t \implies s \in \mathcal{T}_G \mathcal{F} \quad s \equiv_{\mathcal{L}} t \implies t \in \mathcal{T}_G \mathcal{F}$
unfolding *myhill-def* **by** *auto*

lemma *myhill-refl* [*simp*]: $s \in \mathcal{T}_G \mathcal{F} \implies s \equiv_{\mathcal{L}} s$
unfolding *myhill-def* **by** *auto*

lemma *myhill-symmetric*: $s \equiv_{\mathcal{L}} t \implies t \equiv_{\mathcal{L}} s$
unfolding *myhill-def* **by** *auto*

lemma *myhill-trans* [*trans*]:
 $s \equiv_{\mathcal{L}} t \implies t \equiv_{\mathcal{L}} u \implies s \equiv_{\mathcal{L}} u$
unfolding *myhill-def* **by** *auto*

abbreviation *myhill-r* ($\langle MN_{\mathcal{L}} \rangle$) **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* ($\lambda X. \text{gta-der } \mathcal{A} \text{ ' } X$) ($\mathcal{T}_G \mathcal{F} // MN_{\mathcal{L}}$) (**is** *inj-on* $?D ?R$)

proof –

{fix $S T$ **assume** *eq-rel*: $S \in ?R \quad T \in ?R \quad ?D \quad S = ?D \quad T$

have $\bigwedge s t. s \in S \implies t \in T \implies s \equiv_{\mathcal{L}} t$

proof –

fix $s t$ **assume** *mem*: $s \in S \quad t \in T$

then obtain t' **where** *res*: $t' \in T \quad \text{gta-der } \mathcal{A} \quad s = \text{gta-der } \mathcal{A} \quad t'$ **using** *eq-rel(3)*

by (*metis image-iff*)

from *res(1)* *mem* **have** $s \in \mathcal{T}_G \mathcal{F} \quad t \in \mathcal{T}_G \mathcal{F} \quad t' \in \mathcal{T}_G \mathcal{F}$ **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 $\bigwedge s t. s \in S \implies t \in T \implies (s, t) \in MN_{\mathcal{L}}$ **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* = $\lambda X. \text{gta-der } \mathcal{A} \text{ ' } X$
have *image*: ?*D* ' ?*R* $\subseteq \text{Pow } (\text{fset } (\text{fPow } (\mathcal{Q} \mathcal{A})))$ **unfolding** *gta-der-def*
by (*meson PowI fPowI ground-ta-der-states ground-term-of-gterm image-subsetI*)
then have *finite* ($\text{Pow } (\text{fset } (\text{fPow } (\mathcal{Q} \mathcal{A})))$) **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* \times ('*q*, '*f*) *ta*

abbreviation *gtt-rules* **where**

gtt-rules $\mathcal{G} \equiv \text{rules } (\text{fst } \mathcal{G}) \mid \cup \mid \text{rules } (\text{snd } \mathcal{G})$

abbreviation *gtt-eps* **where**

gtt-eps $\mathcal{G} \equiv \text{eps } (\text{fst } \mathcal{G}) \mid \cup \mid \text{eps } (\text{snd } \mathcal{G})$

definition *gtt-states* **where**

gtt-states $\mathcal{G} = \mathcal{Q} (\text{fst } \mathcal{G}) \mid \cup \mid \mathcal{Q} (\text{snd } \mathcal{G})$

abbreviation *gtt-syms* **where**

gtt-syms $\mathcal{G} \equiv \text{ta-sig } (\text{fst } \mathcal{G}) \mid \cup \mid \text{ta-sig } (\text{snd } \mathcal{G})$

definition *gtt-interface* **where**

gtt-interface $\mathcal{G} = \mathcal{Q} (\text{fst } \mathcal{G}) \mid \cap \mid \mathcal{Q} (\text{snd } \mathcal{G})$

definition *gtt-eps-free* **where**

gtt-eps-free $\mathcal{G} = (\text{eps-free } (\text{fst } \mathcal{G}), \text{eps-free } (\text{snd } \mathcal{G}))$

definition *is-gtt-eps-free* :: ('*q*, '*f*) *ta* \times ('*p*, '*g*) *ta* \Rightarrow *bool* **where**

is-gtt-eps-free $\mathcal{G} \longleftrightarrow \text{eps } (\text{fst } \mathcal{G}) = \{\mid\} \wedge \text{eps } (\text{snd } \mathcal{G}) = \{\mid\}$

anchored language accepted by a GTT

definition *agtt-lang* :: ('*q*, '*f*) *gtt* \Rightarrow '*f* *gterm rel* **where**

agtt-lang $\mathcal{G} = \{(t, u) \mid t \text{ u } q. q \mid \in \mid \text{gta-der } (\text{fst } \mathcal{G}) t \wedge q \mid \in \mid \text{gta-der } (\text{snd } \mathcal{G}) u\}$

lemma *agtt-langI*:

$q \mid \in \mid \text{gta-der } (\text{fst } \mathcal{G}) s \Longrightarrow q \mid \in \mid \text{gta-der } (\text{snd } \mathcal{G}) t \Longrightarrow (s, t) \in \text{agtt-lang } \mathcal{G}$

by (auto simp: agtt-lang-def)

lemma *agtt-langE*:

assumes $(s, t) \in \text{agtt-lang } \mathcal{G}$

obtains q where $q \in | \text{gta-der } (\text{fst } \mathcal{G}) s q \in | \text{gta-der } (\text{snd } \mathcal{G}) t$

using *assms* by (auto simp: agtt-lang-def)

lemma *converse-agtt-lang*:

$(\text{agtt-lang } \mathcal{G})^{-1} = \text{agtt-lang } (\text{prod.swap } \mathcal{G})$

by (auto simp: agtt-lang-def)

lemma *agtt-lang-swap*:

$\text{agtt-lang } (\text{prod.swap } \mathcal{G}) = \text{prod.swap } ' \text{agtt-lang } \mathcal{G}$

by (auto simp: agtt-lang-def)

language accepted by a GTT

abbreviation *gtt-lang* :: $('q, 'f) \text{gtt} \Rightarrow 'f \text{gterm rel}$ where

$\text{gtt-lang } \mathcal{G} \equiv \text{gmctxt-cl UNIV } (\text{agtt-lang } \mathcal{G})$

lemma *gtt-lang-join*:

$q \in | \text{gta-der } (\text{fst } \mathcal{G}) s \Longrightarrow q \in | \text{gta-der } (\text{snd } \mathcal{G}) t \Longrightarrow (s, t) \in \text{gmctxt-cl UNIV } (\text{agtt-lang } \mathcal{G})$

by (auto simp: agtt-lang-def)

definition *gtt-accept* where

$\text{gtt-accept } \mathcal{G} s t \equiv (s, t) \in \text{gmctxt-cl UNIV } (\text{agtt-lang } \mathcal{G})$

lemma *gtt-accept-intros*:

$(s, t) \in \text{agtt-lang } \mathcal{G} \Longrightarrow \text{gtt-accept } \mathcal{G} s t$

$\text{length } ss = \text{length } ts \Longrightarrow \forall i < \text{length } ts. \text{gtt-accept } \mathcal{G} (ss ! i) (ts ! i) \Longrightarrow$

$(f, \text{length } ss) \in \mathcal{F} \Longrightarrow \text{gtt-accept } \mathcal{G} (\text{GFun } f ss) (\text{GFun } f ts)$

by (auto simp: gtt-accept-def)

abbreviation *gtt-lang-terms* :: $('q, 'f) \text{gtt} \Rightarrow ('f, 'q) \text{term rel}$ where

$\text{gtt-lang-terms } \mathcal{G} \equiv (\lambda s. \text{map-both term-of-gterm } s) ' (\text{gmctxt-cl UNIV } (\text{agtt-lang } \mathcal{G}))$

lemma *term-of-gterm-gtt-lang-gtt-lang-terms-conv*:

$\text{map-both term-of-gterm } ' \text{gtt-lang } \mathcal{G} = \text{gtt-lang-terms } \mathcal{G}$

by auto

lemma *gtt-accept-swap* [*simp*]:

$\text{gtt-accept } (\text{prod.swap } \mathcal{G}) s t \longleftrightarrow \text{gtt-accept } \mathcal{G} t s$

by (auto simp: gmctxt-cl-swap agtt-lang-swap gtt-accept-def)

lemma *gtt-lang-swap*:

$(\text{gtt-lang } (A, B))^{-1} = \text{gtt-lang } (B, A)$

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 $\exists u. u \in | \text{ta-der}' (\text{fst } \mathcal{G}) (\text{term-of-gterm } s) \wedge u \in | \text{ta-der}' (\text{snd } \mathcal{G}) (\text{term-of-gterm } t)$
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*: $\exists us. \text{length } us = \text{length } ss \wedge$
 $(\forall i < \text{length } ss. (us ! i) \in | \text{ta-der}' (\text{fst } \mathcal{G}) (\text{term-of-gterm } (ss ! i)) \wedge$
 $(us ! i) \in | \text{ta-der}' (\text{snd } \mathcal{G}) (\text{term-of-gterm } (ts ! i)))$
using *Ex-list-of-length-P*[*of length* ss $\lambda u i. u \in | \text{ta-der}' (\text{fst } \mathcal{G}) (\text{term-of-gterm } (ss ! i)) \wedge$
 $u \in | \text{ta-der}' (\text{snd } \mathcal{G}) (\text{term-of-gterm } (ts ! i))$]
by *auto*
then obtain us **where** $\text{length } us = \text{length } ss \wedge (\forall i < \text{length } ss.$
 $(us ! i) \in | \text{ta-der}' (\text{fst } \mathcal{G}) (\text{term-of-gterm } (ss ! i)) \wedge (us ! i) \in | \text{ta-der}'$
 $(\text{snd } \mathcal{G}) (\text{term-of-gterm } (ts ! i)))$
by *blast*
then have $\text{Fun } f \ us \in | \text{ta-der}' (\text{fst } \mathcal{G}) (\text{Fun } f (\text{map } \text{term-of-gterm } ss)) \wedge$
 $\text{Fun } f \ us \in | \text{ta-der}' (\text{snd } \mathcal{G}) (\text{Fun } f (\text{map } \text{term-of-gterm } ts))$ **using** *step(1)*
by *fastforce*
then show *?case* **by** (*metis term-of-gterm.simps*)
qed

lemma *agtt-lang-mono*:
assumes $\text{rules } (\text{fst } \mathcal{G}) \subseteq | \text{rules } (\text{fst } \mathcal{G}') \text{ eps } (\text{fst } \mathcal{G}) \subseteq | \text{eps } (\text{fst } \mathcal{G}')$
 $\text{rules } (\text{snd } \mathcal{G}) \subseteq | \text{rules } (\text{snd } \mathcal{G}') \text{ eps } (\text{snd } \mathcal{G}) \subseteq | \text{eps } (\text{snd } \mathcal{G}')$
shows $\text{agtt-lang } \mathcal{G} \subseteq \text{agtt-lang } \mathcal{G}'$
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 $\text{rules } (\text{fst } \mathcal{G}) \subseteq | \text{rules } (\text{fst } \mathcal{G}') \text{ eps } (\text{fst } \mathcal{G}) \subseteq | \text{eps } (\text{fst } \mathcal{G}')$
 $\text{rules } (\text{snd } \mathcal{G}) \subseteq | \text{rules } (\text{snd } \mathcal{G}') \text{ eps } (\text{snd } \mathcal{G}) \subseteq | \text{eps } (\text{snd } \mathcal{G}')$
shows $\text{gtt-lang } \mathcal{G} \subseteq \text{gtt-lang } \mathcal{G}'$
using *agtt-lang-mono*[*OF assms*]
by (*intro gmctxt-cl-mono-rel*) *auto*

definition *fmap-states-gtt* **where**
 $\text{fmap-states-gtt } f \equiv \text{map-both } (\text{fmap-states-ta } f)$

lemma *ground-map-vars-term-simp*:

ground t \implies *map-term f g t* = *map-term f* ($\lambda\cdot$. *undefined*) *t*
by (*induct t*) *auto*

lemma *states-fmap-states-gtt* [*simp*]:

gtt-states (*fmap-states-gtt f* \mathcal{G}) = *f* |¹ *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}) \cup \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* \subseteq *?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* \subseteq *?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}) $|\subseteq|$ *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 $\mathcal{G} = (\text{let } \text{iface} = \text{gtt-interface } \mathcal{G} \text{ in}$
 $\text{map-both } (\text{ta-only-prod } \text{iface}) \mathcal{G})$

lemma *agtt-only-prod-lang*:
agtt-lang (*gtt-only-prod* \mathcal{G}) = *agtt-lang* \mathcal{G} (**is** $?Ls = ?Rs$)

proof –

let $?A = \text{fst } \mathcal{G}$ **let** $?B = \text{snd } \mathcal{G}$
have $?Ls \subseteq ?Rs$ **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** $(s, t) \in ?Rs$
then obtain q **where** $r: q \in | \text{ta-der } (\text{fst } \mathcal{G}) (\text{term-of-gterm } s) q \in | \text{ta-der}$
 $(\text{snd } \mathcal{G}) (\text{term-of-gterm } t)$
by (*auto simp*: *agtt-lang-def gta-der-def*)
then have $q \in | \text{gtt-interface } \mathcal{G}$ **by** (*auto simp*: *gtt-interface-def*)
then have $(s, t) \in ?Ls$ **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}) $|\subseteq|$ *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* \circ *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*:

$gtt\text{-}syms (trim\text{-}gtt G) \subseteq gtt\text{-}syms G$
unfolding $trim\text{-}gtt\text{-}def$ **using** $fsubset\text{-}trans[OF gtt\text{-}only\text{-}prod\text{-}syms gtt\text{-}only\text{-}reach\text{-}syms]$
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 $ta\text{-}nr\text{-}states :: ('q, 'f) ta \Rightarrow 'q fset$ **where**
 $ta\text{-}nr\text{-}states A = \bigcup | ((fset\text{-}of\text{-}list \circ r\text{-}lhs\text{-}states) |^q | (rules A))$

definition $gtt\text{-}nr\text{-}states$ **where**
 $gtt\text{-}nr\text{-}states G = ta\text{-}nr\text{-}states (fst G) \bigcup | ta\text{-}nr\text{-}states (snd G)$

definition $gtt\text{-}root\text{-}clean$ **where**
 $gtt\text{-}root\text{-}clean G \iff gtt\text{-}nr\text{-}states G \cap | gtt\text{-}interface G = \{\}\}$

4.5 Relabeling

definition $relabel\text{-}gtt :: ('q :: linorder, 'f) gtt \Rightarrow (nat, 'f) gtt$ **where**
 $relabel\text{-}gtt G = fmap\text{-}states\text{-}gtt (map\text{-}fset\text{-}to\text{-}nat (gtt\text{-}states G)) G$

lemma $relabel\text{-}agtt\text{-}lang [simp]$:
 $agtt\text{-}lang (relabel\text{-}gtt G) = agtt\text{-}lang G$
by $(simp add: agtt\text{-}lang\text{-}fmap\text{-}states\text{-}gtt map\text{-}fset\text{-}to\text{-}nat\text{-}inj relabel\text{-}gtt\text{-}def)$

lemma $agtt\text{-}lang\text{-}sig$:
 $fset (gtt\text{-}syms G) \subseteq \mathcal{F} \implies agtt\text{-}lang G \subseteq \mathcal{T}_G \mathcal{F} \times \mathcal{T}_G \mathcal{F}$
by $(auto simp: agtt\text{-}lang\text{-}def gta\text{-}der\text{-}def \mathcal{T}_G\text{-}equivalent\text{-}def)$
 $(metis ffunas\text{-}gterm.rep\text{-}eq less\text{-}eq\text{-}fset.rep\text{-}eq subset\text{-}iff ta\text{-}der\text{-}gterm\text{-}sig)+$

4.6 epsilon free GTTs

lemma $agtt\text{-}lang\text{-}gtt\text{-}eps\text{-}free [simp]$:
 $agtt\text{-}lang (gtt\text{-}eps\text{-}free \mathcal{G}) = agtt\text{-}lang \mathcal{G}$
by $(auto simp: agtt\text{-}lang\text{-}def gta\text{-}der\text{-}def gtt\text{-}eps\text{-}free\text{-}def ta\text{-}res\text{-}eps\text{-}free)$

lemma $gtt\text{-}lang\text{-}gtt\text{-}eps\text{-}free [simp]$:
 $gtt\text{-}lang (gtt\text{-}eps\text{-}free \mathcal{G}) = gtt\text{-}lang \mathcal{G}$
by $auto$

end
theory $GTT\text{-}Compose$
imports GTT
begin

4.7 GTT closure under composition

inductive-set $\Delta_\varepsilon\text{-set} :: ('q, 'f) ta \Rightarrow ('q, 'f) ta \Rightarrow ('q \times 'q) \text{ set for } \mathcal{A} \mathcal{B} \text{ where}$
 $\Delta_\varepsilon\text{-set-cong}: TA\text{-rule } f ps p |\in| \text{ rules } \mathcal{A} \Longrightarrow TA\text{-rule } f qs q |\in| \text{ rules } \mathcal{B} \Longrightarrow \text{length}$
 $ps = \text{length } qs \Longrightarrow$

$(\bigwedge i. i < \text{length } qs \Longrightarrow (ps ! i, qs ! i) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B}) \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B}$
 $|\Delta_\varepsilon\text{-set-eps1}: (p, p') |\in| \text{ eps } \mathcal{A} \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B} \Longrightarrow (p', q) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B}$
 $|\Delta_\varepsilon\text{-set-eps2}: (q, q') |\in| \text{ eps } \mathcal{B} \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B} \Longrightarrow (p, q') \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B}$

lemma $\Delta_\varepsilon\text{-states}: \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B} \subseteq \text{fset } (\mathcal{Q} \mathcal{A} \times \mathcal{Q} \mathcal{B})$

proof –

{fix $p q$ **assume** $(p, q) \in \Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B}$ **then have** $(p, q) \in \text{fset } (\mathcal{Q} \mathcal{A} \times \mathcal{Q} \mathcal{B})$
by (*induct*) (*auto dest: rule-statesD eps-statesD*)
then show *?thesis* **by** *auto*

qed

lemma $\text{finite-}\Delta_\varepsilon \text{ [simp]: finite } (\Delta_\varepsilon\text{-set } \mathcal{A} \mathcal{B})$

using $\text{finite-subset}[OF \Delta_\varepsilon\text{-states}]$

by *simp*

context

includes *fset.lifting*

begin

lift-definition $\Delta_\varepsilon :: ('q, 'f) ta \Rightarrow ('q, 'f) ta \Rightarrow ('q \times 'q) \text{ fset is } \Delta_\varepsilon\text{-set by simp}$

lemmas $\Delta_\varepsilon\text{-cong} = \Delta_\varepsilon\text{-set-cong} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-eps1} = \Delta_\varepsilon\text{-set-eps1} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-eps2} = \Delta_\varepsilon\text{-set-eps2} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-cases} = \Delta_\varepsilon\text{-set.cases} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-induct} [consumes 1, case-names \Delta_\varepsilon\text{-cong } \Delta_\varepsilon\text{-eps1 } \Delta_\varepsilon\text{-eps2}] = \Delta_\varepsilon\text{-set.induct} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-intros} = \Delta_\varepsilon\text{-set.intros} [Transfer.transferred]$

lemmas $\Delta_\varepsilon\text{-simps} = \Delta_\varepsilon\text{-set.simps} [Transfer.transferred]$

end

lemma $\text{finite-alt-def} \text{ [simp]:}$

$\text{finite } \{(\alpha, \beta). (\exists t. \text{ground } t \wedge \alpha |\in| \text{ta-der } \mathcal{A} t \wedge \beta |\in| \text{ta-der } \mathcal{B} t)\} \text{ (is finite ?S)}$

by (*auto dest: ground-ta-der-states[THEN fsubsetD]*)

intro!: $\text{finite-subset}[of ?S \text{fset } (\mathcal{Q} \mathcal{A} \times \mathcal{Q} \mathcal{B})]$

lemma $\Delta_\varepsilon\text{-def}'$:

$\Delta_\varepsilon \mathcal{A} \mathcal{B} = \{(\alpha, \beta). (\exists t. \text{ground } t \wedge \alpha |\in| \text{ta-der } \mathcal{A} t \wedge \beta |\in| \text{ta-der } \mathcal{B} t)\}$

proof (*intro fset-eqI iffI, goal-cases lr rl*)

case (*lr x*) **obtain** $p q$ **where** $x \text{ [simp]: } x = (p, q)$ **by** (*cases x*)

have $\exists t. \text{ground } t \wedge p |\in| \text{ta-der } \mathcal{A} t \wedge q |\in| \text{ta-der } \mathcal{B} t$ **using** *lr unfolding x*

proof (*induct rule: \Delta_\varepsilon-induct*)

case $(\Delta_\varepsilon\text{-cong } f ps p qs q)$

obtain ts **where** $ts: \text{ground } (ts i) \wedge ps ! i |\in| \text{ta-der } \mathcal{A} (ts i) \wedge qs ! i |\in| \text{ta-der } \mathcal{B} (ts i)$

if $i < \text{length } qs$ **for** i **using** $\Delta_\varepsilon\text{-cong}(5)$ **by** *metis*

then show *?case* **using** $\Delta_\varepsilon\text{-cong}(1-3)$

by (*auto intro!*: $\text{exI}[of - Fun f (map ts [0..<length qs])]$) *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** $\text{ground } t\ p \in | \text{ta-der } \mathcal{A}\ t\ q \in | \text{ta-der } \mathcal{B}\ t$ **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' |∈| rules* $\mathcal{A}\ p' = p \vee (p', p) \in | (\text{eps } \mathcal{A})|^{+}$ $\text{length } ps = \text{length } ts$
 $\wedge i. i < \text{length } ts \implies ps\ !\ i \in | \text{ta-der } \mathcal{A}\ (ts\ !\ i)$ **using** *Fun(3) by auto*
obtain $q'\ qs$ **where** q' : $f\ qs \rightarrow q' \in | \text{rules } \mathcal{B}\ q' = q \vee (q', q) \in | (\text{eps } \mathcal{B})|^{+}$
 $\text{length } qs = \text{length } ts$
 $\wedge i. i < \text{length } ts \implies qs\ !\ i \in | \text{ta-der } \mathcal{B}\ (ts\ !\ i)$ **using** *Fun(4) by auto*
have $st: (p', q') \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}$
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 $(p', p) \in | (\text{eps } \mathcal{A})|^{+}$ **then have** $(p, q') \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}$ **using** *st*
by (*induct rule: francl-induct*) (*auto intro:* Δ_ε -*eps1*)**}**
from *st this p'(2)* **have** $st: (p, q') \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}$ **by** *auto*
{assume $(q', q) \in | (\text{eps } \mathcal{B})|^{+}$ **then have** $(p, q) \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}$ **using** *st*
by (*induct rule: francl-induct*) (*auto intro:* Δ_ε -*eps2*)**}**
from *st this q'(2)* **show** $(p, q) \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}$ **by** *auto*
qed *auto*
qed

lemma Δ_ε -*fmember*:

$(p, q) \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} \longleftrightarrow (\exists t. \text{ground } t \wedge p \in | \text{ta-der } \mathcal{A}\ t \wedge q \in | \text{ta-der } \mathcal{B}\ t)$
by (*auto simp:* Δ_ε -*def'*)

definition *GTT-comp* :: (q, f) *gtt* \Rightarrow (q, f) *gtt* \Rightarrow (q, f) *gtt* **where**

$\text{GTT-comp } \mathcal{G}_1\ \mathcal{G}_2 =$
(let $\Delta = \Delta_\varepsilon$ (*snd* \mathcal{G}_1) (*fst* \mathcal{G}_2) *in*
 $(\text{TA } (\text{gtt-rules } (\text{fst } \mathcal{G}_1, \text{fst } \mathcal{G}_2)) (\text{eps } (\text{fst } \mathcal{G}_1) \cup \text{eps } (\text{fst } \mathcal{G}_2) \cup \Delta),$
 $\text{TA } (\text{gtt-rules } (\text{snd } \mathcal{G}_1, \text{snd } \mathcal{G}_2)) (\text{eps } (\text{snd } \mathcal{G}_1) \cup \text{eps } (\text{snd } \mathcal{G}_2) \cup (\Delta^{-1}))$)

lemma *gtt-syms-GTT-comp*:

$\text{gtt-syms } (\text{GTT-comp } A\ B) = \text{gtt-syms } A \cup \text{gtt-syms } B$
by (*auto simp:* *GTT-comp-def ta-sig-def Let-def*)

lemma Δ_ε -*statesD*:

$(p, q) \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} \implies p \in | \mathcal{Q}\ \mathcal{A}$
 $(p, q) \in | \Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} \implies q \in | \mathcal{Q}\ \mathcal{B}$
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'*:

$q \in | \text{eps-states } (\Delta_\varepsilon\ \mathcal{A}\ \mathcal{B}) \implies q \in | \mathcal{Q}\ \mathcal{A} \cup \mathcal{Q}\ \mathcal{B}$
by (*auto simp: eps-states-def dest:* Δ_ε -*statesD*)

lemma Δ_ε -swap:

prod.swap $p \mid \in \mid \Delta_\varepsilon \mathcal{A} \mathcal{B} \longleftrightarrow p \mid \in \mid \Delta_\varepsilon \mathcal{B} \mathcal{A}$
by (*auto simp: Δ_ε -def'*)

lemma Δ_ε -inverse [*simp*]:

$(\Delta_\varepsilon \mathcal{A} \mathcal{B}) \mid^{-1} \mid = \Delta_\varepsilon \mathcal{B} \mathcal{A}$
by (*auto simp: Δ_ε -def'*)

lemma *gtt-states-comp-union*:

gtt-states (*GTT-comp* $\mathcal{G}_1 \mathcal{G}_2$) $\mid \subseteq \mid$ *gtt-states* $\mathcal{G}_1 \mid \cup \mid$ *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 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 $s: q \mid \in \mid$ *gta-der* (*fst* \mathcal{G}_1) s **and** $u: q \mid \in \mid$ *gta-der* (*snd* \mathcal{G}_1) u **and** *ut*:
gtt-accept \mathcal{G}_2 u t

shows $q \mid \in \mid$ *gta-der* (*fst* (*GTT-comp* $\mathcal{G}_1 \mathcal{G}_2$)) s $q \mid \in \mid$ *gta-der* (*snd* (*GTT-comp*
 $\mathcal{G}_1 \mathcal{G}_2$)) t

proof (*goal-cases L R*)

let $?G =$ *GTT-comp* $\mathcal{G}_1 \mathcal{G}_2$

have *sub1l*: *rules* (*fst* \mathcal{G}_1) $\mid \subseteq \mid$ *rules* (*fst* $?G$) *eps* (*fst* \mathcal{G}_1) $\mid \subseteq \mid$ *eps* (*fst* $?G$)

and *sub1r*: *rules* (*snd* \mathcal{G}_1) $\mid \subseteq \mid$ *rules* (*snd* $?G$) *eps* (*snd* \mathcal{G}_1) $\mid \subseteq \mid$ *eps* (*snd* $?G$)

and *sub2r*: *rules* (*snd* \mathcal{G}_2) $\mid \subseteq \mid$ *rules* (*snd* $?G$) *eps* (*snd* \mathcal{G}_2) $\mid \subseteq \mid$ *eps* (*snd* $?G$)

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** $p: p \mid \in \mid$ *gta-der* (*fst* \mathcal{G}_2) s $p \mid \in \mid$ *gta-der* (*snd*
 \mathcal{G}_2) t

by (*auto simp: agtt-lang-def*)

then have $(p, q) \mid \in \mid$ *eps* (*snd* (*GTT-comp* $\mathcal{G}_1 \mathcal{G}_2$))

using Δ_ε -fmember[*of p q fst \mathcal{G}_2 snd \mathcal{G}_1*] *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 $p \mid \in \mid$ *rules* (*snd* \mathcal{G}_1) $p = q$

$\vee (p, q) \in | (eps (snd \mathcal{G}_1))|^+ |$
 $length\ ps = length\ ts \wedge i. i < length\ ts \implies ps ! i \in | gta\text{-}der (snd \mathcal{G}_1) (ss !$
 $i)$
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 \mathcal{G}_2 - -*
prod.swap \mathcal{G}_1 for \mathcal{G}_1 \mathcal{G}_2,
unfolded fst-swap snd-swap GTT-comp-swap gtt-accept-swap]

lemma *gtt-comp-acomplete:*
gcomp-rel UNIV (agtt-lang \mathcal{G}_1) (agtt-lang \mathcal{G}_2) \subseteq agtt-lang (GTT-comp \mathcal{G}_1 \mathcal{G}_2)
proof (*intro subrelI, goal-cases LR*)
case (*LR s t*)
then consider
 $q\ u$ **where** $q \in | gta\text{-}der (fst \mathcal{G}_1) s\ q \in | gta\text{-}der (snd \mathcal{G}_1) u\ gtt\text{-}accept\ \mathcal{G}_2\ u\ t$
 $| q\ u$ **where** $q \in | gta\text{-}der (snd \mathcal{G}_2) t\ q \in | gta\text{-}der (fst \mathcal{G}_2) u\ gtt\text{-}accept\ \mathcal{G}_1\ s\ u$
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 *\Delta_\epsilon\text{-steps-from-}\mathcal{G}_2:*
assumes $(q, q') \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | q \in | gtt\text{-}states\ \mathcal{G}_2$
 $gtt\text{-}states\ \mathcal{G}_1 \cap | gtt\text{-}states\ \mathcal{G}_2 = \{|\}$
shows $(q, q') \in | (eps (fst \mathcal{G}_2))^+ | \wedge q' \in | gtt\text{-}states\ \mathcal{G}_2$
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 \Delta_\epsilon\text{-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 \Delta_\epsilon\text{-statesD}(1)*)
then show *?case* **using** *Step(3)*
by (*auto intro: ftrancl-trans*)
qed

lemma Δ_ε -steps-from- \mathcal{G}_1 :

assumes $(p, r) \in (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))^+ \mid p \in gtt-states \mathcal{G}_1$
 $gtt-states \mathcal{G}_1 \cap gtt-states \mathcal{G}_2 = \{\}$
obtains $r \in gtt-states \mathcal{G}_1 \mid (p, r) \in (eps (fst \mathcal{G}_1))^+ \mid$
 $\mid q p' \textbf{ where } r \in gtt-states \mathcal{G}_2 \mid p = p' \vee (p, p') \in (eps (fst \mathcal{G}_1))^+ \mid (p', q) \in$
 $\Delta_\varepsilon (snd \mathcal{G}_1) (fst \mathcal{G}_2)$
 $q = r \vee (q, r) \in (eps (fst \mathcal{G}_2))^+ \mid$
using *assms(1,2)*
proof (*induct arbitrary: thesis rule: converse-ftrancl-induct*)
case (*Base p*)
from *Base(1)* **consider** (*a*) $(p, r) \in eps (fst \mathcal{G}_1) \mid$ (*b*) $(p, r) \in eps (fst \mathcal{G}_2) \mid$
(*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 $(q, p) \in (eps (fst \mathcal{G}_1))^+ \mid p \in gtt-states \mathcal{G}_1$
 $\mid (q, p) \in \Delta_\varepsilon (snd \mathcal{G}_1) (fst \mathcal{G}_2) \mid p \in gtt-states \mathcal{G}_2$ **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 $(q, q') \in (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))^+ \mid q \in gtt-states \mathcal{G}_1 \mid \cup$
 $gtt-states \mathcal{G}_2$
 $gtt-states \mathcal{G}_1 \cap gtt-states \mathcal{G}_2 = \{\}$
obtains $q \in gtt-states \mathcal{G}_1 \mid q' \in gtt-states \mathcal{G}_1 \mid (q, q') \in (eps (fst \mathcal{G}_1))^+ \mid$
 $\mid p p' \textbf{ where } q \in gtt-states \mathcal{G}_1 \mid q' \in gtt-states \mathcal{G}_2 \mid q = p \vee (q, p) \in (eps (fst$
 $\mathcal{G}_1))^+ \mid$
 $(p, p') \in \Delta_\varepsilon (snd \mathcal{G}_1) (fst \mathcal{G}_2) \mid p' = q' \vee (p', q') \in (eps (fst \mathcal{G}_2))^+ \mid$
 $\mid q \in gtt-states \mathcal{G}_2 \mid (q, q') \in (eps (fst \mathcal{G}_2))^+ \mid \wedge q' \in gtt-states \mathcal{G}_2$
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*:

$(p, q) \in eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) \implies p \in gtt-states \mathcal{G}_1 \mid \cup gtt-states \mathcal{G}_2$
 $(p, q) \in eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) \implies q \in gtt-states \mathcal{G}_1 \mid \cup gtt-states \mathcal{G}_2$

by (auto simp: GTT-comp-def gtt-states-def dest: eps-statesD Δ_ε -statesD)

lemma GTT-comp-eps-ftrancl-fst-statesD:

$(p, q) \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2))) |^+ \implies p \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$

$(p, q) \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2))) |^+ \implies q \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$

using GTT-comp-eps-fst-statesD[of - - $\mathcal{G}_1 \mathcal{G}_2$]

by (meson converse-ftranclE ftranclE)+

lemma GTT-comp-first:

assumes $q \in | ta-der (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) t q \in | gtt-states \mathcal{G}_1$

$gtt-states \mathcal{G}_1 \cap | gtt-states \mathcal{G}_2 = \{ \}$

shows $q \in | ta-der (fst \mathcal{G}_1) t$

using assms(1,2)

proof (induct t arbitrary: q)

case (Var q')

have $q \neq q' \implies q' \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$ 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 $q': TA-rule f qs q' \in | rules (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2))$

$q' = q \vee (q', q) \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2))) |^+ length qs = length ts$

$\wedge i. i < length ts \implies qs ! i \in | ta-der (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) (ts ! i)$

using Fun(2) by auto

have $q' \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$ using $q'(1)$

by (auto simp: GTT-comp-def gtt-states-def dest: rule-statesD)

then have $st: q' \in | gtt-states \mathcal{G}_1$ and $eps: q' = q \vee (q', q) \in | (eps (fst \mathcal{G}_1)) |^+$

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 $q' \in | rules (fst \mathcal{G}_1)$ using assms(3) $q'(1)$

by (auto simp: GTT-comp-def gtt-states-def dest: rule-statesD)

have $i < length ts \implies qs ! i \in | ta-der (fst \mathcal{G}_1) (ts ! i)$ 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  $\mathcal{G}_1 \mathcal{G}_2 :: ('f, 'q) \text{ gtt}$ 
assumes  $as2: \text{gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$ 
and  $1: q \mid \in \mid \text{gta-der } (\text{fst } (\text{GTT-comp } \mathcal{G}_1 \mathcal{G}_2)) \ s \ q \mid \in \mid \text{gta-der } (\text{snd } (\text{GTT-comp } \mathcal{G}_1 \mathcal{G}_2)) \ t \ q \mid \in \mid \text{gtt-states } \mathcal{G}_1$ 
shows  $\exists u. q \mid \in \mid \text{gta-der } (\text{snd } \mathcal{G}_1) \ u \wedge \text{gtt-accept } \mathcal{G}_2 \ u \ t$  using  $1(2,3)$  unfolding
 $\text{gta-der-def}$ 
proof (induct rule: ta-der-gterm-induct)
case ( $\text{GFun } f \ ts \ ps \ p \ q$ )
show  $?case$ 
proof ( $\text{cases } p \mid \in \mid \text{gtt-states } \mathcal{G}_1$ )
case  $\text{True}$ 
then have  $*$ :  $\text{TA-rule } f \ ps \ p \mid \in \mid \text{rules } (\text{snd } \mathcal{G}_1)$  using  $\text{GFun}(1, 6)$   $as2$ 
by ( $\text{auto simp: GTT-comp-def gtt-states-def dest: rule-statesD}$ )
moreover have  $st: i < \text{length } ps \implies ps \ ! \ i \mid \in \mid \text{gtt-states } \mathcal{G}_1$  for  $i$  using  $*$ 
by ( $\text{force simp: gtt-states-def dest: rule-statesD}$ )
moreover have  $i < \text{length } ps \implies \exists u. ps \ ! \ i \mid \in \mid \text{ta-der } (\text{snd } \mathcal{G}_1) (\text{term-of-gterm } u) \wedge \text{gtt-accept } \mathcal{G}_2 \ u (ts \ ! \ i)$  for  $i$ 
using  $st$   $\text{GFun}(2)$  by ( $\text{intro } \text{GFun}(5)$ )  $\text{simp}$ 
then obtain  $us$  where
 $\wedge i. i < \text{length } ps \implies ps \ ! \ i \mid \in \mid \text{ta-der } (\text{snd } \mathcal{G}_1) (\text{term-of-gterm } (us \ i)) \wedge \text{gtt-accept } \mathcal{G}_2 (us \ i) (ts \ ! \ i)$ 
by  $\text{metis}$ 
moreover have  $p = q \vee (p, q) \mid \in \mid (\text{eps } (\text{snd } \mathcal{G}_1)) \mid^+ \mid$  using  $\text{GFun}(3, 6)$   $\text{True}$ 
 $as2$ 
by ( $\text{auto simp: gtt-states-def elim!: } \Delta_\varepsilon\text{-steps-from-}\mathcal{G}_1\text{-}\mathcal{G}_2[\text{of } p \ q \ \text{prod.swap } \mathcal{G}_2 \ \text{prod.swap } \mathcal{G}_1, \ \text{simplified}]$ )
ultimately show  $?thesis$  using  $\text{GFun}(2)$ 
by ( $\text{intro exI}[\text{of } - \ \text{GFun } f \ (\text{map } us \ [0..<\text{length } ts])]$ )
 $(\text{auto simp: gtt-accept-def intro!: exI}[\text{of } - \ ps] \ \text{exI}[\text{of } - \ p])$ 
next
case  $\text{False}$  note  $nt\text{-}st = \text{this}$ 
then have  $\text{False}: p \neq q$  using  $\text{GFun}(6)$  by  $\text{auto}$ 
then have  $\text{eps}: (p, q) \mid \in \mid (\text{eps } (\text{snd } (\text{GTT-comp } \mathcal{G}_1 \mathcal{G}_2))) \mid^+ \mid$  using  $\text{GFun}(3)$ 
by  $\text{simp}$ 
show  $?thesis$  using  $\Delta_\varepsilon\text{-steps-from-}\mathcal{G}_1\text{-}\mathcal{G}_2[\text{of } p \ q \ \text{prod.swap } \mathcal{G}_2 \ \text{prod.swap } \mathcal{G}_1, \ \text{simplified}, \ \text{OF } \text{eps}]$ 
proof ( $\text{cases, goal-cases}$ )
case  $1$  then show  $?case$  using  $\text{False}$   $\text{GFun}(3)$ 
by ( $\text{metis GTT-comp-eps-ftrancl-fst-statesD}(1) \ \text{GTT-comp-swap fst-swap union-iff}$ )
next
case  $2$  then show  $?case$  using  $as2$  by ( $\text{auto simp: gtt-states-def}$ )
next
case  $3$  then show  $?case$  using  $as2$   $\text{GFun}(6)$  by ( $\text{auto simp: gtt-states-def}$ )
next
case  $(4 \ r \ p')$ 
have  $\text{meet}: r \mid \in \mid \text{ta-der } (\text{snd } (\text{GTT-comp } \mathcal{G}_1 \mathcal{G}_2)) (\text{Fun } f \ (\text{map } \text{term-of-gterm } ts))$ 
using  $\text{GFun}(1 - 4)$   $4(3)$   $\text{False}$ 

```

by (*auto simp: GTT-comp-def in-ftrancl-UnI intro!: exI[of - ps] exI[of - p]*)
then obtain u **where** $wit: \text{ground } u \ p' \mid \in \mid \text{ta-der } (\text{snd } \mathcal{G}_1) \ u \ r \mid \in \mid \text{ta-der } (\text{fst } \mathcal{G}_2) \ u$
using $4(4-)$ **unfolding** $\Delta_\varepsilon\text{-def}'$ **by** *blast*
from $wit(1, 3)$ **have** $\text{gtt-accept } \mathcal{G}_2 \ (\text{gterm-of-term } u) \ (G\text{Fun } f \ ts)$
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 $\text{gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$
shows $\text{agtt-lang } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2) \subseteq \text{gcomp-rel UNIV } (\text{agtt-lang } \mathcal{G}_1) \ (\text{agtt-lang } \mathcal{G}_2)$
proof (*intro subrelI, goal-cases LR*)
case (*LR s t*)
obtain q **where** $q: q \mid \in \mid \text{gta-der } (\text{fst } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ s \ q \mid \in \mid \text{gta-der } (\text{snd } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ t$
using *LR* **by** (*auto simp: agtt-lang-def*)
 $\{$
fix $\mathcal{G}_1 \ \mathcal{G}_2 \ s \ t$ **assume** $as2: \text{gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$
and $1: q \mid \in \mid \text{ta-der } (\text{fst } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ (\text{term-of-gterm } s)$
 $q \mid \in \mid \text{ta-der } (\text{snd } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ (\text{term-of-gterm } t) \ q \mid \in \mid \text{gtt-states } \mathcal{G}_1$
note $st = GTT\text{-comp-first}[OF \ 1(1,3) \ as2]$
obtain u **where** $u: q \mid \in \mid \text{ta-der } (\text{snd } \mathcal{G}_1) \ (\text{term-of-gterm } u) \ \text{gtt-accept } \mathcal{G}_2 \ u \ t$
using *gtt-comp-sound-semi[OF as2 1[folded gta-der-def]]* **by** (*auto simp: gta-der-def*)
have $(s, u) \in \text{agtt-lang } \mathcal{G}_1$ **using** $st \ u(1)$
by (*auto simp: agtt-lang-def gta-der-def*)
moreover have $(u, t) \in \text{gtt-lang } \mathcal{G}_2$ **using** $u(2)$
by (*auto simp: gtt-accept-def*)
ultimately have $(s, t) \in \text{agtt-lang } \mathcal{G}_1 \ O \ \text{gmctxt-cl UNIV } (\text{agtt-lang } \mathcal{G}_2)$
by *auto*
note $base = \text{this}$
consider $q \mid \in \mid \text{gtt-states } \mathcal{G}_1 \mid q \mid \in \mid \text{gtt-states } \mathcal{G}_2 \mid q \notin \mid \text{gtt-states } \mathcal{G}_1 \mid \cup \mid \text{gtt-states } \mathcal{G}_2$ **by** *blast*
then show $?case$ **using** $q \ \text{assms}$
proof (*cases, goal-cases*)
case 1 **then show** $?case$ **using** $base[\text{of } \mathcal{G}_1 \ \mathcal{G}_2 \ s \ t]$
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  $\mathcal{G}_2 \subseteq$  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  $\mathcal{G}_1$  | $\cap$ | 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  $\mathcal{G}_1$  | $\cap$ | 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'  $\mathcal{G}_1$   $\mathcal{G}_2 \equiv$  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 $\Delta\text{-trancl-set} :: ('q, 'f) ta \Rightarrow ('q, 'f) ta \Rightarrow ('q \times 'q) \text{ set for } A B$
where

$\Delta\text{-set-cong}$: $TA\text{-rule } f ps p \mid \in \mid \text{ rules } A \Longrightarrow TA\text{-rule } f qs q \mid \in \mid \text{ rules } B \Longrightarrow \text{length } ps = \text{length } qs \Longrightarrow$

$(\bigwedge i. i < \text{length } qs \Longrightarrow (ps ! i, qs ! i) \in \Delta\text{-trancl-set } A B) \Longrightarrow (p, q) \in \Delta\text{-trancl-set } A B$

$\mid \Delta\text{-set-eps1}$: $(p, p') \mid \in \mid \text{ eps } A \Longrightarrow (p, q) \in \Delta\text{-trancl-set } A B \Longrightarrow (p', q) \in \Delta\text{-trancl-set } A B$

$\mid \Delta\text{-set-eps2}$: $(q, q') \mid \in \mid \text{ eps } B \Longrightarrow (p, q) \in \Delta\text{-trancl-set } A B \Longrightarrow (p, q') \in \Delta\text{-trancl-set } A B$

$\mid \Delta\text{-set-trans}$: $(p, q) \in \Delta\text{-trancl-set } A B \Longrightarrow (q, r) \in \Delta\text{-trancl-set } A B \Longrightarrow (p, r) \in \Delta\text{-trancl-set } A B$

lemma $\Delta\text{-trancl-set-states}$: $\Delta\text{-trancl-set } \mathcal{A} \mathcal{B} \subseteq \text{fset } (\mathcal{Q} \mathcal{A} \mid \times \mid \mathcal{Q} \mathcal{B})$

proof –

{fix $p q$ **assume** $(p, q) \in \Delta\text{-trancl-set } \mathcal{A} \mathcal{B}$ **then have** $(p, q) \in \text{fset } (\mathcal{Q} \mathcal{A} \mid \times \mid \mathcal{Q} \mathcal{B})$

by $(\text{induct}) (\text{auto dest: rule-statesD eps-statesD})$

then show $?thesis$ **by auto**

qed

lemma $\text{finite-}\Delta\text{-trancl-set}$ $[\text{simp}]$: $\text{finite } (\Delta\text{-trancl-set } \mathcal{A} \mathcal{B})$

using $\text{finite-subset}[OF \Delta\text{-trancl-set-states}]$

by simp

context

includes fset.lifting

begin

lift-definition $\Delta\text{-trancl} :: ('q, 'f) ta \Rightarrow ('q, 'f) ta \Rightarrow ('q \times 'q) \text{ fset is } \Delta\text{-trancl-set}$
by simp

lemmas $\Delta\text{-trancl-cong} = \Delta\text{-set-cong}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-eps1} = \Delta\text{-set-eps1}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-eps2} = \Delta\text{-set-eps2}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-cases} = \Delta\text{-trancl-set.cases}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-induct}$ $[\text{consumes } 1, \text{case-names } \Delta\text{-cong } \Delta\text{-eps1 } \Delta\text{-eps2 } \Delta\text{-trans}]$
 $= \Delta\text{-trancl-set.induct}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-intros} = \Delta\text{-trancl-set.intros}$ $[\text{Transfer.transferred}]$

lemmas $\Delta\text{-trancl-simps} = \Delta\text{-trancl-set.simps}$ $[\text{Transfer.transferred}]$

end

lemma $\Delta\text{-trancl-cl}$ $[\text{simp}]$:

$(\Delta\text{-trancl } A B)^{+} = \Delta\text{-trancl } A B$

proof –

{fix $s t$ **assume** $(s, t) \mid \in \mid (\Delta\text{-trancl } A B)^{+}$ **then have** $(s, t) \mid \in \mid \Delta\text{-trancl } A B$
by $(\text{induct rule: ftrancl-induct}) (\text{auto intro: } \Delta\text{-trancl-intros})$

then show $?thesis$ **by auto**

qed

lemma Δ -trancl-states: Δ -trancl $\mathcal{A} \mathcal{B} \mid \subseteq \mid (\mathcal{Q} \mathcal{A} \mid \times \mid \mathcal{Q} \mathcal{B})$
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 $\Delta = \Delta$ -trancl (snd G) (fst G) in
 (TA (rules (fst G)) (eps (fst G) $\mid \cup \mid \Delta$),
 TA (rules (snd G)) (eps (snd G) $\mid \cup \mid (\Delta \mid^{-1} \mid))))$

lemma Δ -trancl-inv:

$(\Delta$ -trancl $A B \mid)^{-1} \mid = \Delta$ -trancl $B A$

proof –

have [*dest*]: $(p, q) \mid \in \mid \Delta$ -trancl $A B \implies (q, p) \mid \in \mid \Delta$ -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) $\mid \subseteq \mid$ *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 $G \subseteq$ *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)) \subseteq *gtt-lang* (*GTT-trancl* G)

proof –

have [*dest*]: $(p, q) \mid \in \mid \Delta_\varepsilon$ (TA (rules A) (eps A $\mid \cup \mid (\Delta$ -trancl $B A$)))
 (TA (rules B) (eps B $\mid \cup \mid (\Delta$ -trancl $A B$))) $\implies (p, q) \mid \in \mid \Delta$ -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 $G \subseteq$ *agtt-lang* (*GTT-trancl* G)

by (rule agtt-lang-mono) (auto simp: GTT-trancl-def Δ -trancl-inv)

lemma Δ_ε -tr-incl:

$\Delta_\varepsilon (TA (rules A) (eps A \mid\cup\mid \Delta\text{-trancl } B A)) (TA (rules B) (eps B \mid\cup\mid \Delta\text{-trancl } A B)) = \Delta\text{-trancl } A B$
(is ?LS = ?RS)

proof –

{fix p q assume (p, q) $\mid\in\mid$?LS then have (p, q) $\mid\in\mid$?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) $\mid\in\mid$?RS then have (p, q) $\mid\in\mid$?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)) \subseteq agtt-lang (GTT-trancl G)

proof –

have [intro!, dest]: (p, q) $\mid\in\mid$ $\Delta_\varepsilon (TA (rules A) (eps A \mid\cup\mid (\Delta\text{-trancl } B A))) (TA (rules B) (eps B \mid\cup\mid (\Delta\text{-trancl } A B))) \implies (p, q) \mid\in\mid \Delta\text{-trancl } A B$ 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) \subseteq 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)⁺ \subseteq agtt-lang (GTT-trancl G)

using *GTT-trancl-base subset-trans*[*OF gtt-comp-lang-complete GTT-trancl-trans*]
by (*metis trancl-id trancl-mono-subset trans-O-iff*)

lemma *trancl-gtt-lang-arg-closed*:
assumes $\text{length } ss = \text{length } ts \ \forall i < \text{length } ts. (ss ! i, ts ! i) \in (\text{gtt-lang } \mathcal{G})^+$
shows $(\text{GFun } f \ ss, \text{GFun } f \ ts) \in (\text{gtt-lang } \mathcal{G})^+ \ (\text{is } ?e \in -)$
proof –
have *all-ctxt-closed-gterm UNIV* $((\text{gtt-lang } \mathcal{G})^+)$ **by** (*intro all-ctxt-closed-gterm-trancl*)
auto
from *all-ctxt-closed-gtermD*[*OF this - assms*] **show** *?thesis*
by *auto*
qed

lemma Δ -*trancl-sound*:
assumes $(p, q) \mid \in \Delta\text{-trancl } A \ B$
obtains $s \ t$ **where** $(s, t) \in (\text{gtt-lang } (B, A))^+ \ p \mid \in \text{gta-der } A \ s \ q \mid \in \text{gta-der } B \ t$
using *assms*
proof (*induct arbitrary: thesis rule: Δ -trancl-induct*)
case $(\Delta\text{-cong } f \ ps \ p \ qs \ q)$
have $\exists si \ ti. (si, ti) \in (\text{gtt-lang } (B, A))^+ \wedge ps ! i \mid \in \text{gta-der } A \ (si) \wedge$
 $qs ! i \mid \in \text{gta-der } B \ (ti)$ **if** $i < \text{length } qs$ **for** i
using $\Delta\text{-cong}(5)$ [*OF that*] **by** *metis*
then obtain $ss \ ts$ **where**
 $\bigwedge i. i < \text{length } qs \implies (ss \ i, ts \ i) \in (\text{gtt-lang } (B, A))^+ \wedge ps ! i \mid \in \text{gta-der } A \ (ss$
 $i) \wedge qs ! i \mid \in \text{gta-der } B \ (ts \ i)$ **by** *metis*
then show *?case* **using** $\Delta\text{-cong}(1-5)$
by (*intro $\Delta\text{-cong}(6)$ [of $\text{GFun } f \ (\text{map } ss \ [0..<\text{length } ps]) \ \text{GFun } f \ (\text{map } ts$*
[$0..<\text{length } qs$]]])
(auto simp: gta-der-def intro!: trancl-gtt-lang-arg-closed)
next
case $(\Delta\text{-eps1 } p \ p' \ q)$ **then show** *?case*
by (*metis gta-der-def ta-der-eps*)
next
case $(\Delta\text{-eps2 } q \ q' \ p)$ **then show** *?case*
by (*metis gta-der-def ta-der-eps*)
next
case $(\Delta\text{-trans } p \ q \ r)$
obtain $s1 \ t1$ **where** $(s1, t1) \in (\text{gtt-lang } (B, A))^+ \ p \mid \in \text{gta-der } A \ s1 \ q \mid \in \text{gta-der}$
 $B \ t1$
using $\Delta\text{-trans}(2)$ **.note** $1 = \text{this}$
obtain $s2 \ t2$ **where** $(s2, t2) \in (\text{gtt-lang } (B, A))^+ \ q \mid \in \text{gta-der } A \ s2 \ r \mid \in \text{gta-der}$
 $B \ t2$
using $\Delta\text{-trans}(4)$ **.note** $2 = \text{this}$
have $(t1, s2) \in \text{gtt-lang } (B, A)$ **using** $1(1,3) \ 2(1,2)$
by (*auto simp: Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang[symmetric] gtt-lang-join*)
then have $(s1, t2) \in (\text{gtt-lang } (B, A))^+ \ \text{using } 1(1) \ 2(1)$
by (*meson trancl.trancl-into-trancl trancl-trans*)
then show *?case* **using** $1(2) \ 2(3)$ **by** (*auto intro: $\Delta\text{-trans}(5)$ [of $s1 \ t2]$*)
qed

lemma *GTT-trancl-sound-aux*:

assumes $p \in | \text{gta-der } (TA \text{ (rules } A) \text{ (eps } A \cup | (\Delta\text{-trancl } B \ A))) \text{ } s$

shows $\exists t. (s, t) \in (\text{gtt-lang } (A, B))^+ \wedge p \in | \text{gta-der } A \ t$

using *assms*

proof (*induct s arbitrary: p*)

case (*GFun f ss*)

let $?eps = \text{eps } A \cup | \Delta\text{-trancl } B \ A$

obtain $qs \ q$ **where** $q: TA\text{-rule } f \ qs \ q \in | \text{rules } A \ q = p \vee (q, p) \in | ?eps|^+ \ \text{length}$
 $qs = \text{length } ss$

$\wedge i. i < \text{length } ss \implies qs \ ! \ i \in | \text{gta-der } (TA \text{ (rules } A) \ ?eps) \ (ss \ ! \ i)$

using *GFun(2)* **by** (*auto simp: gta-der-def*)

have $\wedge i. i < \text{length } ss \implies \exists ti. (ss \ ! \ i, ti) \in (\text{gtt-lang } (A, B))^+ \wedge qs \ ! \ i \in |$
 $\text{gta-der } A \ (ti)$

using *GFun(1)[OF nth-mem q(4)]* **unfolding** *gta-der-def* **by** *fastforce*

then obtain ts **where** $ts: \wedge i. i < \text{length } ss \implies (ss \ ! \ i, ts \ i) \in (\text{gtt-lang } (A,$
 $B))^+ \wedge qs \ ! \ i \in | \text{gta-der } A \ (ts \ i)$

by *metis*

then have $q': q \in | \text{gta-der } A \ (GFun \ f \ (\text{map } ts \ [0..<\text{length } ss]))$
 $(GFun \ f \ ss, GFun \ f \ (\text{map } ts \ [0..<\text{length } ss])) \in (\text{gtt-lang } (A, B))^+ \ \text{using } q(1,$
 $3)$

by (*auto simp: gta-der-def intro!: exI[of - qs] exI[of - q] trancl-gtt-lang-arg-closed*)

{fix $p \ q \ u$ **assume** $ass: (p, q) \in | \Delta\text{-trancl } B \ A \ (GFun \ f \ ss, u) \in (\text{gtt-lang } (A,$
 $B))^+ \wedge p \in | \text{gta-der } A \ u$

from $\Delta\text{-trancl-sound}[OF \ \text{this}(1)]$ **obtain** $s \ t$

where $(s, t) \in (\text{gtt-lang } (A, B))^+ \ p \in | \text{gta-der } B \ s \ q \in | \text{gta-der } A \ t \ . \ \text{note}$
 $st = \text{this}$

have $(u, s) \in \text{gtt-lang } (A, B)$ **using** *st conjunct2[OF ass(2)]*

by (*auto simp: Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang[symmetric] gtt-lang-join*)

then have $(GFun \ f \ ss, t) \in (\text{gtt-lang } (A, B))^+$

using *ass st(1)* **by** (*meson trancl-into-trancl2 trancl-trans*)

then have $\exists s \ t. (GFun \ f \ ss, t) \in (\text{gtt-lang } (A, B))^+ \wedge q \in | \text{gta-der } A \ t$ **using**
 $st \ \text{by } \text{blast}$

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 $(q, p) \in | ?eps|^+ \ \text{using } q(2) \ \text{by } \text{simp}$

then show *?thesis using q(1) q'*

proof (*induct rule: ftrancl-induct*)

case (*Base q p*) **from** *Base(1)* **show** *?case*

proof

assume $(q, p) \in | \text{eps } A$ **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])]*)

$trancl\text{-}gtt\text{-}lang\text{-}arg\text{-}closed\ exI[of - qs]\ exI[of - q]$

```

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 |∈|

```

$gta\text{-}der\ (snd\ (GTT\text{-}trancl\ G))\ t$
by (*auto simp: agtt-lang-def*)
obtain s' **where** $(s, s') \in (gtt\text{-}lang\ G)^+ q \mid \in \mid gta\text{-}der\ (fst\ G)\ s'$ **using** *base join*
by (*auto simp: GTT-trancl-def Δ -trancl-inv agtt-lang-def*)
moreover obtain t' **where** $(t', t) \in (gtt\text{-}lang\ G)^+ q \mid \in \mid gta\text{-}der\ (snd\ G)\ t'$
using *join*
by (*auto simp: GTT-trancl-def gtt-lang-swap[*of fst G snd G, symmetric*]*
trancl-converse Δ -trancl-inv)
moreover have $(s', t') \in gtt\text{-}lang\ G$ **using** *calculation*
by (*auto simp: Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang[symmetric] gtt-lang-join*)
ultimately show $(s, t) \in (gtt\text{-}lang\ G)^+$ **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*:
 $agtt\text{-}lang\ (GTT\text{-}trancl\ G) = gtrancl\text{-}rel\ UNIV\ (agtt\text{-}lang\ G)$
using *GTT-trancl-asound GTT-trancl-acomplete* **by** *blast*

lemma *GTT-trancl-lang*:
 $gtt\text{-}lang\ (GTT\text{-}trancl\ G) = (gtt\text{-}lang\ G)^+$
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* :: $('q, 'f)\ gtt \Rightarrow ('q \times 'q)\ fset \Rightarrow 'f\ gterm\ rel$ **where**
 $pair\text{-}at\text{-}lang\ \mathcal{G}\ Q = \{(s, t) \mid s\ t\ p\ q.\ q \mid \in \mid gta\text{-}der\ (fst\ \mathcal{G})\ s \wedge p \mid \in \mid gta\text{-}der\ (snd\ \mathcal{G})\ t \wedge (q, p) \mid \in \mid Q\}$

lemma *pair-at-lang-restr-states*:
 $pair\text{-}at\text{-}lang\ \mathcal{G}\ Q = pair\text{-}at\text{-}lang\ \mathcal{G}\ (Q \mid \cap \mid (Q\ (fst\ \mathcal{G}) \mid \times \mid Q\ (snd\ \mathcal{G})))$
by (*auto simp: pair-at-lang-def gta-der-def*) (*meson gterm-ta-der-states*)

lemma *pair-at-langE*:
assumes $(s, t) \in pair\text{-}at\text{-}lang\ \mathcal{G}\ Q$
obtains $q\ p$ **where** $(q, p) \mid \in \mid Q$ **and** $q \mid \in \mid gta\text{-}der\ (fst\ \mathcal{G})\ s$ **and** $p \mid \in \mid gta\text{-}der\ (snd\ \mathcal{G})\ t$
using *assms* **by** (*auto simp: pair-at-lang-def*)

lemma *pair-at-langI*:
assumes $q \mid \in \mid gta\text{-}der\ (fst\ \mathcal{G})\ s\ p \mid \in \mid gta\text{-}der\ (snd\ \mathcal{G})\ t\ (q, p) \mid \in \mid Q$
shows $(s, t) \in pair\text{-}at\text{-}lang\ \mathcal{G}\ Q$
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 $\mathcal{Q} \sqsubseteq \mathcal{Q}$ (*fst* \mathcal{G}) \times \mathcal{Q} (*snd* \mathcal{G})
shows *pair-at-lang* \mathcal{G} $\mathcal{Q} =$ *pair-at-lang* (*map-prod* (*fmap-states-ta f*) (*fmap-states-ta g*) \mathcal{G}) (*map-prod f g* \upharpoonright \mathcal{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* \in *ta-der* (*fst* \mathcal{G}) (*term-of-gterm s*) *f p* \in *ta-der* (*fmap-states-ta f* (*fst* \mathcal{G})) (*term-of-gterm s*) **and**
ts: q \in *ta-der* (*snd* \mathcal{G}) (*term-of-gterm t*) *g q* \in *ta-der* (*fmap-states-ta g* (*snd* \mathcal{G})) (*term-of-gterm t*) **and**
st: (f p, g q) \in (*map-prod f g* \upharpoonright \mathcal{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* \in \mathcal{Q} (*fst* \mathcal{G}) *q* \in \mathcal{Q} (*snd* \mathcal{G}) **by** *auto*
then have (*p, q*) \in \mathcal{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} \mathcal{Q})⁻¹ = *pair-at-lang* (*prod.swap* \mathcal{G}) (\mathcal{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 \mathcal{Q}_1 \mathcal{G}_2 $\mathcal{Q}_2 \equiv \mathcal{Q}_1 \mid \mathcal{O} \mid \Delta_\varepsilon$ (*snd* \mathcal{G}_1) (*fst* \mathcal{G}_2) $\mid \mathcal{O} \mid \mathcal{Q}_2$

lemma *pair-comp-sound1*:
assumes (*s, t*) \in *pair-at-lang* \mathcal{G}_1 \mathcal{Q}_1
and (*t, u*) \in *pair-at-lang* \mathcal{G}_2 \mathcal{Q}_2

shows $(s, u) \in \text{pair-at-lang } (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2) (\Delta\text{-eps-pair } \mathcal{G}_1 \ Q_1 \ \mathcal{G}_2 \ Q_2)$
proof –
from $\text{pair-at-langE } \text{assms}$ **obtain** $p \ q \ q' \ r$ **where**
 $\text{wit: } (p, q) \in Q_1 \ p \in \text{gta-der } (\text{fst } \mathcal{G}_1) \ s \ q \in \text{gta-der } (\text{snd } \mathcal{G}_1) \ t$
 $(q', r) \in Q_2 \ q' \in \text{gta-der } (\text{fst } \mathcal{G}_2) \ t \ r \in \text{gta-der } (\text{snd } \mathcal{G}_2) \ u$
by metis
from $\text{wit}(3, 5)$ **have** $(q, q') \in \Delta_\varepsilon (\text{snd } \mathcal{G}_1) (\text{fst } \mathcal{G}_2)$
by $(\text{auto simp: } \Delta_\varepsilon\text{-def' gta-der-def intro!: exI[of - term-of-gterm t])$
then have $(p, r) \in \Delta\text{-eps-pair } \mathcal{G}_1 \ Q_1 \ \mathcal{G}_2 \ Q_2$ **using** $\text{wit}(1, 4)$
by $(\text{auto simp: } \Delta\text{-eps-pair-def})$
then show $?thesis$ **using** $\text{wit}(2, 6)$ **unfolding** pair-at-lang-def
by auto
qed

lemma pair-comp-sound2 :
assumes $(s, u) \in \text{pair-at-lang } (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2) (\Delta\text{-eps-pair } \mathcal{G}_1 \ Q_1 \ \mathcal{G}_2 \ Q_2)$
shows $\exists t. (s, t) \in \text{pair-at-lang } \mathcal{G}_1 \ Q_1 \wedge (t, u) \in \text{pair-at-lang } \mathcal{G}_2 \ Q_2$
using assms **unfolding** $\text{pair-at-lang-def } \Delta\text{-eps-pair-def}$
by $(\text{auto simp: } \Delta_\varepsilon\text{-def' gta-der-def}) (\text{metis gterm-of-term-inv})$

lemma pair-comp-sound :
 $\text{pair-at-lang } \mathcal{G}_1 \ Q_1 \ O \ \text{pair-at-lang } \mathcal{G}_2 \ Q_2 = \text{pair-at-lang } (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2) (\Delta\text{-eps-pair } \mathcal{G}_1 \ Q_1 \ \mathcal{G}_2 \ Q_2)$
by $(\text{auto simp: } \text{pair-comp-sound1 } \text{pair-comp-sound2 } \text{relcomp.simps})$

inductive-set $\Delta\text{-Atrans-set} :: ('q \times 'q) \text{fset} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q \times 'q) \text{set}$ **for** $Q \ \mathcal{A} \ \mathcal{B}$ **where**
 $\text{base [simp]: } (p, q) \in Q \Longrightarrow (p, q) \in \Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B}$
 $\text{| step [intro]: } (p, q) \in \Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B} \Longrightarrow (q, r) \in \Delta_\varepsilon \ \mathcal{B} \ \mathcal{A} \Longrightarrow$
 $(r, v) \in \Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B} \Longrightarrow (p, v) \in \Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B}$

lemma $\Delta\text{-Atrans-set-states}$:
 $(p, q) \in \Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B} \Longrightarrow (p, q) \in \text{fset } ((\text{fst } | \cdot | \ Q \ |\cup| \ Q \ \mathcal{A}) \ |\times| \ (\text{snd } | \cdot | \ Q \ |\cup| \ Q \ \mathcal{B}))$
by $(\text{induct rule: } \Delta\text{-Atrans-set.induct}) (\text{auto simp: image-iff intro!: bexI})$

lemma $\text{finite-}\Delta\text{-Atrans-set}$: $\text{finite } (\Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B})$

proof –
have $\Delta\text{-Atrans-set } Q \ \mathcal{A} \ \mathcal{B} \subseteq \text{fset } ((\text{fst } | \cdot | \ Q \ |\cup| \ Q \ \mathcal{A}) \ |\times| \ (\text{snd } | \cdot | \ Q \ |\cup| \ Q \ \mathcal{B}))$
using $\Delta\text{-Atrans-set-states}$
by (metis subrelI)
from $\text{finite-subset[OF this]}$ **show** $?thesis$ **by** simp
qed

context

includes fset.lifting

begin

lift-definition $\Delta\text{-Atrans} :: ('q \times 'q) \text{fset} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q \times 'q) \text{fset}$ **is** $\Delta\text{-Atrans-set}$

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} $Q \equiv \Delta$ -Atrans Q (*fst* \mathcal{G}) (*snd* \mathcal{G})

lemma *pair-trancl-sound1*:
assumes $(s, t) \in (\text{pair-at-lang } \mathcal{G} \ Q)^+$
shows $\exists q \ p. \ p \ |\in| \text{gta-der } (\text{fst } \mathcal{G}) \ s \wedge q \ |\in| \text{gta-der } (\text{snd } \mathcal{G}) \ t \wedge (p, q) \ |\in| \Delta$ -Atrans-gtt $\mathcal{G} \ Q$
using *assms*
proof (*induct*)
case (*step t v*)
obtain $p \ q \ r \ r'$ where *reach-t*: $r \ |\in| \text{gta-der } (\text{fst } \mathcal{G}) \ t \ q \ |\in| \text{gta-der } (\text{snd } \mathcal{G}) \ t$
and
reach: $p \ |\in| \text{gta-der } (\text{fst } \mathcal{G}) \ s \ r' \ |\in| \text{gta-der } (\text{snd } \mathcal{G}) \ v$ and
st: $(p, q) \ |\in| \Delta$ -Atrans-gtt $\mathcal{G} \ Q$ $(r, r') \ |\in| Q$ using *step(2, 3)*
by (*auto simp: pair-at-lang-def*)
from *reach-t* have $(q, r) \ |\in| \Delta_\varepsilon (\text{snd } \mathcal{G}) (\text{fst } \mathcal{G})$
by (*auto simp: Δ_ε -def' gta-der-def intro: ground-term-of-gterm*)
then have $(p, r') \ |\in| \Delta$ -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 $(p, q) \ |\in| \Delta$ -Atrans-gtt $\mathcal{G} \ Q$
and $p \ |\in| \text{gta-der } (\text{fst } \mathcal{G}) \ s \ q \ |\in| \text{gta-der } (\text{snd } \mathcal{G}) \ t$
shows $(s, t) \in (\text{pair-at-lang } \mathcal{G} \ Q)^+$ 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*:
 $(\text{pair-at-lang } \mathcal{G} \ Q)^+ = \text{pair-at-lang } \mathcal{G} \ (\Delta$ -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* $\mathcal{A} \ |\cup| (\text{fId-on } (Q \ \mathcal{A}) \ |O| \ Q)$)
definition *pair-at-to-agtt* :: $('q, 'f) \text{gtt} \Rightarrow ('q \times 'q) \text{fset} \Rightarrow ('q, 'f) \text{gtt}$ where
pair-at-to-agtt $\mathcal{G} \ Q = (\text{fst-pair-cl } (\text{fst } \mathcal{G}) \ Q, TA$ (*rules* (*snd* \mathcal{G})) (*eps* (*snd* \mathcal{G})))

lemma *fst-pair-cl-eps*:

assumes $(p, q) \in (eps (fst\text{-}pair\text{-}cl \mathcal{A} Q))^{+|}$
and $\mathcal{Q} \mathcal{A} \mid \cap \mid snd \mid \uparrow Q = \{\mid\}$
shows $(p, q) \in (eps \mathcal{A})^{+|} \vee (\exists r. (p = r \vee (p, r) \in (eps \mathcal{A})^{+|}) \wedge (r, q) \in Q)$ **using** *assms*

proof (*induct rule: ftrancl-induct*)
case (*Step p q r*)
then have $y: q \in \mathcal{Q} \mathcal{A}$ **by** (*auto simp add: eps-trancl-statesD eps-statesD*)
have $[simp]: (p, q) \in Q \implies q \in snd \mid \uparrow Q$ **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 $\mathcal{Q} \mathcal{A} \mid \cap \mid snd \mid \uparrow Q = \{\mid\}$
and $q \in ta\text{-}der (fst\text{-}pair\text{-}cl \mathcal{A} Q) (term\text{-}of\text{-}gterm \ t)$
shows $\exists p. p \in ta\text{-}der \mathcal{A} (term\text{-}of\text{-}gterm \ t) \wedge (q \notin \mathcal{Q} \mathcal{A} \longrightarrow (p, q) \in Q) \wedge (q \in \mathcal{Q} \mathcal{A} \longrightarrow p = q)$ **using** *assms*

proof (*induct t arbitrary: q*)
case (*GFun f ts*)
then obtain $qs \ q'$ **where** *rule: TA-rule f qs q' \in rules \mathcal{A} length qs = length ts*
and
 $eps: q' = q \vee (q', q) \in (eps (fst\text{-}pair\text{-}cl \mathcal{A} Q))^{+|}$ **and**
 $reach: \forall i < length \ ts. qs \ ! \ i \in ta\text{-}der (fst\text{-}pair\text{-}cl \mathcal{A} Q) (term\text{-}of\text{-}gterm \ (ts \ ! \ i))$
by *auto*
{fix i **assume** $ass: i < length \ ts$ **then have** $st: qs \ ! \ i \in \mathcal{Q} \mathcal{A}$ **using** *rule*
by (*auto simp: rule-statesD*)
then have $qs \ ! \ i \notin snd \mid \uparrow Q$ **using** *GFun(2)* **by** *auto*
then have $qs \ ! \ i \in ta\text{-}der \ \mathcal{A} (term\text{-}of\text{-}gterm \ (ts \ ! \ i))$ **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** $nt\text{-}eq = this$
then have $eps: (q', q) \in (eps (fst\text{-}pair\text{-}cl \mathcal{A} Q))^{+|}$ **using** *eps* **by** *simp*
from *fst-pair-cl-eps[OF this assms(1)]* **show** *?thesis*
using *False rule IH*
proof (*cases q \notin \mathcal{Q} \mathcal{A}*)
case *True*
from *fst-pair-cl-eps[OF eps assms(1)]* **obtain** r **where**
 $q' = r \vee (q', r) \in (eps \ \mathcal{A})^{+|}$ $(r, q) \in Q$ **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) ∈ ?LS by (auto simp: agtt-lang-def pair-at-to-agtt-def gta-der-def
      intro: pair-at-langI)}
  then show ?RS ⊆ ?LS by auto
qed

```

definition *pair-at-to-agtt'* **where**

pair-at-to-agtt' $\mathcal{G} \ Q = (\text{let } \mathcal{A} = \text{fmap-states-ta Inl } (\text{fst } \mathcal{G}) \text{ in}$
 $\text{let } \mathcal{B} = \text{fmap-states-ta Inr } (\text{snd } \mathcal{G}) \text{ in}$
 $\text{let } Q' = Q \mid \cap \mid (Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})) \text{ in}$
 $\text{pair-at-to-agtt } (\mathcal{A}, \mathcal{B}) \ (\text{map-prod Inl Inr } \mid \uparrow \mid Q')$)

lemma *pair-at-agtt-cost*:

pair-at-lang $\mathcal{G} \ Q = \text{agtt-lang } (\text{pair-at-to-agtt}' \ \mathcal{G} \ Q)$

proof –

let $?G = (\text{fmap-states-ta CInl } (\text{fst } \mathcal{G}), \text{fmap-states-ta CInr } (\text{snd } \mathcal{G}))$
let $?Q = (Q \mid \cap \mid (Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})))$
let $?Q' = \text{map-prod CInl CInr } \mid \uparrow \mid ?Q$
have $*$: *pair-at-lang* $\mathcal{G} \ Q = \text{pair-at-lang } \mathcal{G} \ ?Q$
using *pair-at-lang-restr-states* **by** *blast*
have *pair-at-lang* $\mathcal{G} \ ?Q = \text{pair-at-lang } (\text{map-prod } (\text{fmap-states-ta CInl}) \ (\text{fmap-states-ta CInr}) \ \mathcal{G}) \ (\text{map-prod CInl CInr } \mid \uparrow \mid ?Q)$
by (*intro pair-at-lang-fun-states*[**where** $?G = \mathcal{G}$ **and** $?Q = ?Q$ **and** $?f = \text{CInl}$
and $?g = \text{CInr}$])
(auto simp: finj-CInl-CInr)
then have $**$: *pair-at-lang* $\mathcal{G} \ ?Q = \text{pair-at-lang } ?G \ ?Q'$ **by** (*simp add: map-prod-simp'*)
have *pair-at-lang* $?G \ ?Q' = \text{agtt-lang } (\text{pair-at-to-agtt}' \ ?G \ ?Q')$
by (*intro pair-at-agtt-conv*[**where** $?G = ?G$]) *auto*
then show *?thesis unfolding * ** pair-at-to-agtt'-def Let-def*
by *simp*

qed

lemma Δ -*Atrans-states-stable*:

assumes $Q \mid \subseteq \mid Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$
shows Δ -*Atrans-gtt* $\mathcal{G} \ Q \mid \subseteq \mid Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$

proof

fix s **assume** $\text{ass}: s \mid \in \mid \Delta$ -*Atrans-gtt* $\mathcal{G} \ Q$
then obtain $t \ u$ **where** $s: s = (t, u)$ **by** (*cases s*) *blast*
show $s \mid \in \mid Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$ **using** ass *assms* **unfolding** s
by (*induct rule: \Delta-Atrans-induct*) *auto*

qed

lemma Δ -*Atrans-map-prod*:

assumes *finj-on* $f \ (Q \ (\text{fst } \mathcal{G}))$ **and** *finj-on* $g \ (Q \ (\text{snd } \mathcal{G}))$
and $Q \mid \subseteq \mid Q \ (\text{fst } \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$
shows *map-prod* $f \ g \ \mid \uparrow \mid (\Delta$ -*Atrans-gtt* $\mathcal{G} \ Q) = \Delta$ -*Atrans-gtt* $(\text{map-prod } (\text{fmap-states-ta } f) \ (\text{fmap-states-ta } g) \ \mathcal{G}) \ (\text{map-prod } f \ g \ \mid \uparrow \mid Q)$
(is $?LS = ?RS)$

proof –

{**fix** $p \ q$ **assume** $(p, q) \mid \in \mid \Delta$ -*Atrans-gtt* $\mathcal{G} \ Q$
then have $(f \ p, g \ q) \mid \in \mid ?RS$ **using** *assms*
proof (*induct rule: \Delta-Atrans-induct*)
case (*step p q r v*)
from *step(3, 6, 7)* **have** $(g \ q, f \ r) \mid \in \mid \Delta_\varepsilon \ (\text{fmap-states-ta } g \ (\text{snd } \mathcal{G}))$
 $(\text{fmap-states-ta } f \ (\text{fst } \mathcal{G}))$

```

    by (auto simp:  $\Delta_\varepsilon$ -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)  $\in$  ?RS
 then have (p, q)  $\in$  ?LS using assms
 proof (induct rule:  $\Delta$ -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:  $\Delta_\varepsilon$  (snd G) (fst G)  $\subseteq$  Q (snd G)  $\times$  Q (fst G)
     using  $\Delta_\varepsilon$ -statesD(1, 2) by fastforce
   have s-e: (?f p, ?g q)  $\in$   $\Delta$ -Atrans-gtt G Q (?f r, ?g v)  $\in$   $\Delta$ -Atrans-gtt G Q
     using step assms(1, 2) fsubsetD[OF  $\Delta$ -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)  $\in$   $\Delta_\varepsilon$  (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:  $\Delta_\varepsilon$ -fmember fimage-iff fBex-def)
     (metis ground-term-of-gterm ground-term-to-gtermD ta-der-fmap-states-inv)
   then have (q, r)  $\in$  map-prod g f  $\uparrow$   $\Delta_\varepsilon$  (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)  $\Delta_\varepsilon$ -statesD(1)  $\Delta_\varepsilon$ -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  $\Delta$ -Atrans-states-stable[OF assms(3)]]
     using  $\Delta$ -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.premis(3) by force
   qed auto}
ultimately show ?thesis by auto
qed

```

— Section: Pair Automaton is closed under Determinization

definition Q -pow where

$$Q\text{-pow } Q \mathcal{S}_1 \mathcal{S}_2 = \{ |(Wrapp X, Wrapp Y) | X Y p q. X \in fPow \mathcal{S}_1 \wedge Y \in fPow \mathcal{S}_2 \wedge p \in X \wedge q \in Y \wedge (p, q) \in Q | \}$$

lemma Q -pow-fmember:

$$(X, Y) \in Q\text{-pow } Q \mathcal{S}_1 \mathcal{S}_2 \iff (\exists p q. ex X \in fPow \mathcal{S}_1 \wedge ex Y \in fPow \mathcal{S}_2 \wedge p \in X \wedge q \in Y \wedge (p, q) \in Q)$$

proof –
let $?S = \{(Wrapp\ X, Wrapp\ Y) \mid X\ Y\ p\ q.\ X \in fPow\ S_1 \wedge Y \in fPow\ S_2 \wedge p \in X \wedge q \in Y \wedge (p, q) \in Q\}$
have $?S \subseteq map\text{-}prod\ Wrapp\ Wrapp\ 'fset\ (fPow\ S_1 \times fPow\ S_2)$ **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 $\mathcal{G}\ Q = pair\text{-}at\text{-}lang\ (map\text{-}both\ ps\text{-}ta\ \mathcal{G})\ (Q\text{-}pow\ Q\ (Q\ (fst\ \mathcal{G}))\ (Q\ (snd\ \mathcal{G})))$ **(is** $?LS = ?RS$ **)**

proof –
{fix $s\ t$ **assume** $(s, t) \in ?LS$
then obtain $p\ q$ **where**
 $res : p \in ta\text{-}der\ (fst\ \mathcal{G})\ (term\text{-}of\text{-}gterm\ s)$
 $q \in ta\text{-}der\ (snd\ \mathcal{G})\ (term\text{-}of\text{-}gterm\ t)\ (p, q) \in Q$
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 $(s, t) \in ?RS$ **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** $(s, t) \in ?RS$ **then have** $(s, t) \in ?LS$
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* $(\mathcal{A}, \mathcal{B})\ (Q\ \mathcal{A} \times Q\ \mathcal{B} \mid\!-\! Q) = gterms\ (fset\ \mathcal{F}) \times gterms\ (fset\ \mathcal{F}) - pair\text{-}at\text{-}lang\ (\mathcal{A}, \mathcal{B})\ Q$
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 $\mathcal{G}_1 \mathcal{G}_2 \equiv (ta\text{-union } (fst \mathcal{G}_1) (fst \mathcal{G}_2),$
 $ta\text{-union } (snd \mathcal{G}_1) (snd \mathcal{G}_2))$

abbreviation *AGTT-union'* **where**

AGTT-union' $\mathcal{G}_1 \mathcal{G}_2 \equiv AGTT\text{-union } (fmap\text{-states-gtt } Inl \mathcal{G}_1) (fmap\text{-states-gtt } Inr \mathcal{G}_2)$

lemma *disj-gtt-states-disj-fst-ta-states*:

assumes *dist-st*: $gtt\text{-states } \mathcal{G}_1 \mid \cap \mid gtt\text{-states } \mathcal{G}_2 = \{\mid\}$
shows $\mathcal{Q} (fst \mathcal{G}_1) \mid \cap \mid \mathcal{Q} (fst \mathcal{G}_2) = \{\mid\}$
using *assms* **unfolding** *gtt-states-def* **by** *auto*

lemma *disj-gtt-states-disj-snd-ta-states*:

assumes *dist-st*: $gtt\text{-states } \mathcal{G}_1 \mid \cap \mid gtt\text{-states } \mathcal{G}_2 = \{\mid\}$
shows $\mathcal{Q} (snd \mathcal{G}_1) \mid \cap \mid \mathcal{Q} (snd \mathcal{G}_2) = \{\mid\}$
using *assms* **unfolding** *gtt-states-def* **by** *auto*

lemma *ta-der-not-contains-undefined-state*:

assumes $q \notin \mathcal{Q} T$ **and** *ground* t
shows $q \notin ta\text{-der } T t$
using *ground-ta-der-states*[*OF* *assms*(2)] *assms*(1)
by *blast*

lemma *AGTT-union-sound1*:

assumes *dist-st*: $gtt\text{-states } \mathcal{G}_1 \mid \cap \mid gtt\text{-states } \mathcal{G}_2 = \{\mid\}$
shows $agtt\text{-lang } (AGTT\text{-union } \mathcal{G}_1 \mathcal{G}_2) \subseteq agtt\text{-lang } \mathcal{G}_1 \cup agtt\text{-lang } \mathcal{G}_2$

proof –

let $?TA\text{-}A = ta\text{-union } (fst \mathcal{G}_1) (fst \mathcal{G}_2)$
let $?TA\text{-}B = ta\text{-union } (snd \mathcal{G}_1) (snd \mathcal{G}_2)$
{fix $s t$ **assume** *ass*: $(s, t) \in agtt\text{-lang } (AGTT\text{-union } \mathcal{G}_1 \mathcal{G}_2)$
then obtain q **where** $ls: q \in ta\text{-der } ?TA\text{-}A$ (*term-of-gterm* s) **and**
 $rs: q \in ta\text{-der } ?TA\text{-}B$ (*term-of-gterm* t)
by (*auto simp add: AGTT-union-def agtt-lang-def gta-der-def*)
then have $(s, t) \in agtt\text{-lang } \mathcal{G}_1 \vee (s, t) \in agtt\text{-lang } \mathcal{G}_2$
proof (*cases* $q \in gtt\text{-states } \mathcal{G}_1$)
case *True*
then have $q \notin gtt\text{-states } \mathcal{G}_2$ **using** *dist-st*
by *blast*
then have *nt-fst-st*: $q \notin \mathcal{Q} (fst \mathcal{G}_2)$ **and**
nt-snd-state: $q \notin \mathcal{Q} (snd \mathcal{G}_2)$ **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  $q \notin \text{gtt-states } \mathcal{G}_1$  by (metis IntI dist-st emptyE)
  then have  $\text{nt-fst-st}: q \notin \mathcal{Q}$  (fst } \mathcal{G}_1) and
     $\text{nt-snd-state}: q \notin \mathcal{Q}$  (snd } \mathcal{G}_1) 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 $\text{agtt-lang } \mathcal{G}_1 \subseteq \text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2)$
 $\text{agtt-lang } \mathcal{G}_2 \subseteq \text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2)$
 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 $\text{dist-st}: \text{gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$
 shows $\text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cup \text{agtt-lang } \mathcal{G}_2$
 using *AGTT-union-sound1*[*OF assms*] *AGTT-union-sound2* by *blast*

lemma *AGTT-union'-sound*:
 fixes $\mathcal{G}_1 :: ('q, 'f) \text{gtt}$ and $\mathcal{G}_2 :: ('q, 'f) \text{gtt}$
 shows $\text{agtt-lang } (\text{AGTT-union}' \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cup \text{agtt-lang } \mathcal{G}_2$
proof –
 have *map*: $\text{agtt-lang } (\text{AGTT-union}' \mathcal{G}_1 \mathcal{G}_2) =$
 $\text{agtt-lang } (\text{fmap-states-gtt } \text{CInl } \mathcal{G}_1) \cup \text{agtt-lang } (\text{fmap-states-gtt } \text{CInr } \mathcal{G}_2)$
 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* $:: ('q, 'f) \text{gtt} \Rightarrow ('q, 'f) \text{gtt} \Rightarrow ('q, 'f) \text{gtt}$ **where**
 $\text{AGTT-comp } \mathcal{G}_1 \mathcal{G}_2 = (\text{let } (\mathcal{A}, \mathcal{B}) = (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2) \text{ in}$
 $(\text{TA } (\text{rules } \mathcal{A}) (\text{eps } \mathcal{A} \mid \cup \mid (\Delta_\epsilon (\text{snd } \mathcal{G}_1) (\text{fst } \mathcal{G}_2) \mid \cap \mid (\text{gtt-interface } \mathcal{G}_1 \mid \times \mid$
 $\text{gtt-interface } \mathcal{G}_2))),$
 $\text{TA } (\text{rules } \mathcal{B}) (\text{eps } \mathcal{B})))$

abbreviation *AGTT-comp'* **where**
 $\text{AGTT-comp}' \mathcal{G}_1 \mathcal{G}_2 \equiv \text{AGTT-comp } (\text{fmap-states-gtt } \text{Inl } \mathcal{G}_1) (\text{fmap-states-gtt } \text{Inr } \mathcal{G}_2)$

lemma *AGTT-comp-sound*:

assumes *gtt-states* $\mathcal{G}_1 \mid \cap \mid$ *gtt-states* $\mathcal{G}_2 = \{\mid\}$
shows *agtt-lang* (*AGTT-comp* $\mathcal{G}_1 \mathcal{G}_2$) = *agtt-lang* \mathcal{G}_1 *O* *agtt-lang* \mathcal{G}_2
proof –
let $?Q_1 = \text{fId-on}$ (*gtt-interface* \mathcal{G}_1) **let** $?Q_2 = \text{fId-on}$ (*gtt-interface* \mathcal{G}_2)
have *lan*: *agtt-lang* $\mathcal{G}_1 = \text{pair-at-lang}$ \mathcal{G}_1 $?Q_1$ *agtt-lang* $\mathcal{G}_2 = \text{pair-at-lang}$ \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* $\mathcal{G}_2 = \text{pair-at-lang}$ (*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* $\mathcal{G}_1 \mathcal{G}_2 = \text{pair-at-to-agtt}$ (*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 -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* \Rightarrow '*b* + '*c*) \mathcal{G}_1 *fmap-states-gtt* (*Inr* :: '*c* \Rightarrow '*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* \Rightarrow ('*q* + '*q*, '*f*) *gtt* **where**

AGTT-trancl $\mathcal{G} = (\text{let } \mathcal{A} = \text{fmap-states-ta } \text{Inl } (\text{fst } \mathcal{G}) \text{ in}$
TA (*rules* \mathcal{A}) (*eps* \mathcal{A} $\mid \cup \mid$ *map-prod* *CInl* *CInr* $\mid \uparrow \mid$ (Δ -*Atrans-gtt* \mathcal{G} (*fId-on* (*gtt-interface* \mathcal{G}))))),
TA (*map-ta-rule* *CInr* *id* $\mid \uparrow \mid$ (*rules* (*snd* \mathcal{G}))) (*map-both* *CInr* $\mid \uparrow \mid$ (*eps* (*snd* \mathcal{G}))))))

lemma *AGTT-trancl-sound*:

shows *agtt-lang* (*AGTT-trancl* \mathcal{G}) = (*agtt-lang* \mathcal{G})⁺

proof –

let $?P = \text{map-prod}$ (*fmap-states-ta* *CInl*) (*fmap-states-ta* *CInr*) \mathcal{G}
let $?Q = \text{fId-on}$ (*gtt-interface* \mathcal{G}) **let** $?Q' = \text{map-prod}$ *CInl* *CInr* $\mid \uparrow \mid$ $?Q$
have *inv*: *finj-on* *CInl* ($?Q$ (*fst* \mathcal{G})) *finj-on* *CInr* ($?Q$ (*snd* \mathcal{G}))
 $?Q \mid \subseteq \mid$ $?Q$ (*fst* \mathcal{G}) $\mid \times \mid$ $?Q$ (*snd* \mathcal{G})
by (*auto simp: gtt-interface-def finj-CInl-CInr*)
have $*$: *fst* $\mid \uparrow \mid$ *map-prod* *CInl* *CInr* $\mid \uparrow \mid$ Δ -*Atrans-gtt* \mathcal{G} (*fId-on* (*gtt-interface* \mathcal{G}))
 $\mid \subseteq \mid$ *CInl* $\mid \uparrow \mid$ $?Q$ (*fst* \mathcal{G})
using *fsubsetD*[*OF* Δ -*Atrans-states-stable*[*OF* *inv*(β)]]
by (*auto simp add: gtt-interface-def*)

from *pair-at-lang-fun-states*[*OF inv*]
have *agtt-lang* $\mathcal{G} = \text{pair-at-lang } ?P ?Q'$ **using** *pair-at-agtt*[*of* \mathcal{G}] **by** *auto*
moreover then have $(\text{agtt-lang } \mathcal{G})^+ = \text{pair-at-lang } ?P (\Delta\text{-Atrans-gtt } ?P ?Q')$
by (*simp add: pair-trancl-sound*)
moreover have *AGTT-trancl* $\mathcal{G} = \text{pair-at-to-agtt } ?P (\Delta\text{-Atrans-gtt } ?P ?Q')$
using $\Delta\text{-Atrans-states-stable}$ [*OF inv*(β)] $\Delta\text{-Atrans-map-prod}$ [*OF inv, symmetric*]
ric]
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* $(\text{map-prod } (\text{fmap-states-ta } \text{CInl}) (\text{fmap-states-ta } \text{CInr}) \mathcal{G}) = \{\|\}$
by (*auto simp: gtt-interface-def*)
ultimately show *?thesis* **using** *pair-at-agtt-conv*[*of* $\Delta\text{-Atrans-gtt } ?P ?Q' ?P$]
 $\Delta\text{-Atrans-states-stable}$ [*OF inv*(β)]
unfolding $\Delta\text{-Atrans-map-prod}$ [*OF inv, symmetric*]
by (*simp add: fimage-mono gtt-interface-def map-prod-ftimes*)
qed

4.12 Anchord gtt intersection

definition *AGTT-inter* **where**

$$\text{AGTT-inter } \mathcal{G}_1 \mathcal{G}_2 \equiv (\text{prod-ta } (\text{fst } \mathcal{G}_1) (\text{fst } \mathcal{G}_2), \\ \text{prod-ta } (\text{snd } \mathcal{G}_1) (\text{snd } \mathcal{G}_2))$$

lemma *AGTT-inter-sound*:

$$\text{agtt-lang } (\text{AGTT-inter } \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cap \text{agtt-lang } \mathcal{G}_2 \text{ (is } ?Ls = ?Rs)$$

proof –

$$\text{let } ?TA\text{-}A = \text{prod-ta } (\text{fst } \mathcal{G}_1) (\text{fst } \mathcal{G}_2)$$

$$\text{let } ?TA\text{-}B = \text{prod-ta } (\text{snd } \mathcal{G}_1) (\text{snd } \mathcal{G}_2)$$

$$\{\text{fix } s \ t \ \text{assume } \text{ass}: (s, t) \in \text{agtt-lang } (\text{AGTT-inter } \mathcal{G}_1 \mathcal{G}_2)$$

$$\text{then obtain } q \ \text{where } ls: q \in | \text{ta-der } ?TA\text{-}A \text{ (term-of-gterm } s) \ \text{and}$$

$$rs: q \in | \text{ta-der } ?TA\text{-}B \text{ (term-of-gterm } t)$$

$$\text{by (auto simp add: AGTT-inter-def agtt-lang-def gta-der-def)}$$

$$\text{then have } (s, t) \in \text{agtt-lang } \mathcal{G}_1 \wedge (s, t) \in \text{agtt-lang } \mathcal{G}_2$$

$$\text{using } \text{prod-ta-der-to-}\mathcal{A}\text{-}\mathcal{B}\text{-der1}[\text{of } q] \ \text{prod-ta-der-to-}\mathcal{A}\text{-}\mathcal{B}\text{-der2}[\text{of } q]$$

$$\text{by (auto simp: agtt-lang-def gta-der-def) blast}\}$$

$$\text{then have } f: ?Ls \subseteq ?Rs \text{ by auto}$$

$$\text{moreover have } ?Rs \subseteq ?Ls \text{ using } \mathcal{A}\text{-}\mathcal{B}\text{-der-to-prod-ta}$$

$$\text{by (fastforce simp: agtt-lang-def AGTT-inter-def gta-der-def)}$$

$$\text{ultimately show } ?thesis \text{ by blast}$$

qed

4.13 Anchord gtt trimming

abbreviation *trim-agtt* \equiv *trim-gtt*

lemma *agtt-only-prod-lang*:

$$\text{agtt-lang } (\text{gtt-only-prod } \mathcal{G}) = \text{agtt-lang } \mathcal{G} \text{ (is } ?Ls = ?Rs)$$

```

proof –
  let ?A = fst  $\mathcal{G}$  let ?B = snd  $\mathcal{G}$ 
  have ?Ls  $\subseteq$  ?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)  $\in$  ?Rs
    then obtain q where r: q  $\in$  ta-der (fst  $\mathcal{G}$ ) (term-of-gterm s) q  $\in$  ta-der
    (snd  $\mathcal{G}$ ) (term-of-gterm t)
    by (auto simp: agtt-lang-def gta-der-def)
    then have q  $\in$  gtt-interface  $\mathcal{G}$  by (auto simp: gtt-interface-def)
    then have (s, t)  $\in$  ?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  $\Rightarrow$  'g gterm  $\Rightarrow$  ('f option  $\times$  'g option) gterm where
  gpair s t = glabel ( $\lambda p$ . (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  $\Rightarrow$  'b list  $\Rightarrow$  ('a option  $\times$  '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 (λx. (Some x, None)) xs
zip-fill [] ys = map (λy. (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 (λf. (Some f, None)) s
| gpair-impl None     (Some t) = map-gterm (λf. (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* \Rightarrow '*f gterm option* **where**

gcollapse (GFun None -) = None
| *gcollapse (GFun (Some f) ts) = Some (GFun f (map the (filter ($\lambda t. \neg Option.is-none$ t) (map gcollapse ts))))*

lemma *gcollapse-groot-None* [*simp*]:

groot-sym t = None \implies gcollapse t = None
fst (groot t) = None \implies gcollapse t = None
by (*cases t, simp*) $+$

definition *gfst* :: ('*f option* \times '*g option*) *gterm* \Rightarrow '*f gterm* **where**

gfst = the \circ gcollapse \circ map-gterm fst

definition *gsnd* :: ('*f option* \times '*g option*) *gterm* \Rightarrow '*g gterm* **where**

gsnd = the \circ gcollapse \circ map-gterm snd

lemma *filter-less-upt*:

$[i \leftarrow [i..<m] \mid i < n] = [i..<\min n m]$

proof (*cases i \leq 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 $gposs\ s = \{p. p \in gposs\ t \wedge gfun-at\ t\ p \neq Some\ None\}$

shows $gposs\ (the\ (gcollapse\ t)) = gposs\ s$

$\bigwedge p. p \in gposs\ s \implies gfun-at\ (the\ (gcollapse\ t))\ p = (gfun-at\ t\ p \gg= id)$

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  $\implies \neg$  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 $\leftarrow$ map gcollapse ts .  $\neg$  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.. $\text{length } ts$ ], unfolded set-upt atLeast-
LessThan-iff]
intro!: nth-equalityI)
have [simp]: i < length ss  $\implies$  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*:
 $map-gterm (the \circ snd) (gpair-impl\ None\ (Some\ t)) = t$
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 $\bigwedge i. i < length\ ts \implies ts\ !\ i = gpair\ (ss\ !\ i)\ (us\ !\ i)$
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 $gpair\ t\ t = map-gterm\ (\lambda f. (Some\ f,\ Some\ f))\ t$
proof (*induct t*)
case ($GFun\ f\ ts$)
then show ?*case* **by** (*auto intro!: gpair-context2[symmetric]*)
qed

lemma *gpair-inject [simp]*:
 $gpair\ s\ t = gpair\ s'\ t' \iff s = s' \wedge t = t'$
by (*metis gfst-gpair gsnd-gpair*)

abbreviation *gterm-to-None-Some* :: $'f\ gterm \Rightarrow ('f\ option \times 'f\ option)\ gterm$
where

$gterm-to-None-Some\ t \equiv map-gterm\ (\lambda f. (None,\ Some\ f))\ t$
abbreviation *gterm-to-Some-None* $t \equiv map-gterm\ (\lambda f. (Some\ f,\ None))\ t$

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 $zip-fill\ ss\ ts = zip\ (map\ Some\ ss)\ (map\ Some\ (take\ (length\ ss)\ ts))\ @\ map\ (\lambda x. (None,\ Some\ x))\ (drop\ (length\ ss)\ ts)$
using *assms by (auto simp: zip-fill-def list-eq-iff-nth-eq nth-append simp add:*

min.absorb2)

lemma *zip-fill2*:

assumes $\text{length } ts < \text{length } ss$
shows $\text{zip-fill } ss \ ts = \text{zip } (\text{map } \text{Some } (\text{take } (\text{length } ts) \ ss)) \ (\text{map } \text{Some } ts) \ @$
 $\text{map } (\lambda x. (\text{Some } x, \text{None})) \ (\text{drop } (\text{length } ts) \ ss)$
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 $p \notin \text{gposs } t$
shows $p \ @ \ q \in \text{gposs } t = \text{False}$ **using** *assms poss-gposs-conv*
using *poss-append-poss* **by** *blast*

lemma *gfun-at-gpair*:

$\text{gfun-at } (\text{gpair } s \ t) \ p = (\text{if } p \in \text{gposs } s \ \text{then } (\text{if } p \in \text{gposs } t$
 $\text{then } \text{Some } (\text{gfun-at } s \ p, \text{gfun-at } t \ p)$
 $\text{else } \text{Some } (\text{gfun-at } s \ p, \text{None})) \ \text{else}$
 $(\text{if } p \in \text{gposs } t \ \text{then } \text{Some } (\text{None}, \text{gfun-at } t \ p) \ \text{else } \text{None}))$
using *gfun-at-glabel* **by** (*auto simp: gpair-def*)

lemma *gposs-of-gpair* [*simp*]:

shows $\text{gposs } (\text{gpair } s \ t) = \text{gposs } s \ \cup \ \text{gposs } t$
by (*auto simp: gpair-def*)

lemma *poss-to-gpair-poss*:

$p \in \text{gposs } s \implies p \in \text{gposs } (\text{gpair } s \ t)$
 $p \in \text{gposs } t \implies p \in \text{gposs } (\text{gpair } s \ t)$
by *auto*

lemma *gsubt-at-gpair-poss*:

assumes $p \in \text{gposs } s$ **and** $p \in \text{gposs } t$
shows $\text{gsubt-at } (\text{gpair } s \ t) \ p = \text{gpair } (\text{gsubt-at } s \ p) \ (\text{gsubt-at } t \ p)$ **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 $p \in \text{gposs } s$ **and** $p \notin \text{gposs } t$
shows $\text{gsubt-at } (\text{gpair } s \ t) \ p = \text{gterm-to-Some-None } (\text{gsubt-at } s \ p)$ **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 $p \in \text{gposs } t$ **and** $p \notin \text{gposs } s$
shows $\text{gsubt-at } (\text{gpair } s \ t) \ p = \text{gterm-to-None-Some } (\text{gsubt-at } t \ p)$ **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** $p \equiv \text{ghole-pos } C$

assumes $p \notin \text{gposs } s$ **and** $\text{gpair } s \ t = C \langle \text{gterm-to-None-Some } u \rangle_G$

shows $\text{gpair } s \ (\text{gctxt-at-pos } t \ p) \langle v \rangle_G = C \langle \text{gterm-to-None-Some } v \rangle_G$

using *assms(2-)*

proof –

note $p[\text{simp}] = \text{assms}(1)$

have $pt: p \in \text{gposs } t$ **and** $pc: p \in \text{gposs } C \langle \text{gterm-to-None-Some } v \rangle_G$

and $pu: p \in \text{gposs } C \langle \text{gterm-to-None-Some } u \rangle_G$

using *arg-cong[OF assms(3), of gposs] assms(2) ghole-pos-in-apply*

by *auto*

have $*$: $\text{gctxt-at-pos } (\text{gpair } s \ (\text{gctxt-at-pos } t \ (\text{ghole-pos } C)) \langle v \rangle_G) \ (\text{ghole-pos } C) =$
 $\text{gctxt-at-pos } (\text{gpair } s \ t) \ (\text{ghole-pos } C)$

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 $\text{gsubt-at } (\text{gpair } s \ (\text{gctxt-at-pos } t \ p) \langle v \rangle_G) \ p = \text{gsubt-at } C \langle \text{gterm-to-None-Some}$
 $v \rangle_G \ p$

using pt *assms(2) subst-at-gpair-nt-poss-None-Some[OF - assms(2), of (gctxt-at-pos*
 $t \ p) \langle v \rangle_G]$

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]*:

$\text{fst } (\text{groot } (\text{gpair } s \ t)) = (\text{Some } (\text{fst } (\text{groot } s)), \text{Some } (\text{fst } (\text{groot } t)))$

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 $\text{adapt-vars-ctxt } (\text{adapt-vars-ctxt } C) = \text{adapt-vars-ctxt } C$ **using** *assms*

by (*induct C*) (*auto simp: adapt-vars2*)

5.4 Encoding of lists of terms

definition *gencode* :: '*f gterm list* \Rightarrow '*f option list gterm* **where**

$gencode\ ts = glabel\ (\lambda p. map\ (\lambda t. gfun-at\ t\ p)\ ts)\ (gunions\ (map\ gdomain\ ts))$

definition $gdecode-nth :: 'f\ option\ list\ gterm \Rightarrow nat \Rightarrow 'f\ gterm$ **where**
 $gdecode-nth\ t\ i = the\ (gcollapse\ (map-gterm\ (\lambda f. f\ !\ i)\ t))$

lemma $gdecode-nth-gencode$:

assumes $i < length\ ts$

shows $gdecode-nth\ (gencode\ ts)\ i = ts\ !\ i$

proof –

have $*$: $gposs\ (ts\ !\ i) = \{p \in gposs\ (map-gterm\ (\lambda f. f\ !\ i)\ (gencode\ ts)).$
 $gfun-at\ (map-gterm\ (\lambda f. f\ !\ i)\ (gencode\ ts))\ p \neq Some\ None\}$

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 $gdecode :: 'f\ option\ list\ gterm \Rightarrow 'f\ gterm\ list$ **where**

$gdecode\ t = (case\ t\ of\ GFun\ f\ ts \Rightarrow map\ (\lambda i. gdecode-nth\ t\ i)\ [0..<length\ f])$

lemma $gdecode-gencode$:

$gdecode\ (gencode\ ts) = ts$

proof $(cases\ gencode\ ts)$

case $(GFun\ f\ ts')$

have $length\ f = length\ ts$ **using** $arg-cong$ $[OF\ GFun, of\ \lambda t. gfun-at\ t\ []]$

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 $gencode-impl :: 'f\ gterm\ option\ list \Rightarrow 'f\ option\ list\ gterm$ **where**

$gencode-impl\ ts = glabel\ (\lambda p. map\ (\lambda t. t \ggg (\lambda t. gfun-at\ t\ p))\ ts)\ (gunions\ (map\ (case-option\ (GFun\ ()\ [])\ gdomain)\ ts))$

lemma $gencode-code$ $[code]$:

$gencode\ ts = gencode-impl\ (map\ Some\ ts)$

by $(auto\ simp: gencode-def\ gencode-impl-def\ comp-def)$

lemma $gencode-singleton$:

$gencode\ [t] = map-gterm\ (\lambda f. [Some\ f])\ t$

using $glabel-map-gterm-conv$ $[unfolded\ comp-def, of\ \lambda t. [t]\ t]$

by $(simp\ add: gunions-def\ gencode-def)$

lemma $gencode-pair$:

$gencode\ [t, u] = map-gterm\ (\lambda (f, g). [f, g])\ (gpair\ t\ u)$

by (*simp add: gunions-def gencode-def gpair-def map-gterm-glabel comp-def*)

5.5 RRn relations

definition *RR1-spec* where

RR1-spec $A T \longleftrightarrow \mathcal{L} A = T$

definition *RR2-spec* where

RR2-spec $A T \longleftrightarrow \mathcal{L} A = \{gpair\ t\ u \mid t\ u. (t, u) \in T\}$

definition *RRn-spec* where

RRn-spec $n\ A\ R \longleftrightarrow \mathcal{L} A = gencode\ 'R \wedge (\forall ts \in R. length\ ts = n)$

lemma *RR1-to-RRn-spec*:

assumes *RR1-spec* $A\ T$

shows *RRn-spec* 1 (*fmap-funs-reg* ($\lambda f. [Some\ f]$) A) ($(\lambda t. [t])\ 'T$)

proof –

have [*simp*]: *inj-on* ($\lambda f. [Some\ f]$) X **for** X **by** (*auto simp: inj-on-def*)

show *?thesis* **using** *assms*

by (*auto simp: RR1-spec-def RRn-spec-def fmap-funs- \mathcal{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* ($\lambda(f, g). [f, g]$) A) ($(\lambda(t, u). [t, u])\ 'T$)

proof –

have [*simp*]: *inj-on* ($\lambda(f, g). [f, g]$) X **for** X **by** (*auto simp: inj-on-def*)

show *?thesis* **using** *assms*

by (*auto simp: RR2-spec-def RRn-spec-def fmap-funs- \mathcal{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* ($\lambda f. (f\ !\ 0, f\ !\ 1)$) A) ($(\lambda f. (f\ !\ 0, f\ !\ 1))\ 'T$) (is *RR2-spec* ? A ? T)

proof –

{**fix** xs **assume** $xs \in T$ **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 $**$: ($\lambda f. (f\ !\ 0, f\ !\ Suc\ 0)$) \circ ($\lambda(f, g). [f, g]$) = *id* **by** *auto*

have *map-gterm* ($\lambda f. (f\ !\ 0, f\ !\ Suc\ 0)$) (*gencode* xs) = *gpair* $t\ u$

unfolding $*$ *gencode-pair gterm.map-comp ** gterm.map-id ..*

then have $\exists t\ u. xs = [t, u] \wedge$ *map-gterm* ($\lambda f. (f\ !\ 0, f\ !\ Suc\ 0)$) (*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]:
 $RR1\text{-spec } (relabel\text{-reg } A) T \longleftrightarrow RR1\text{-spec } A T$
 by (simp add: RR1-spec-def)

lemma *relabel-RR2-spec* [simp]:
 $RR2\text{-spec } (relabel\text{-reg } A) T \longleftrightarrow RR2\text{-spec } A T$
 by (simp add: RR2-spec-def)

lemma *relabel-RRn-spec* [simp]:
 $RRn\text{-spec } n (relabel\text{-reg } A) T \longleftrightarrow RRn\text{-spec } n A T$
 by (simp add: RRn-spec-def)

lemma *trim-RR1-spec* [simp]:
 $RR1\text{-spec } (trim\text{-reg } A) T \longleftrightarrow RR1\text{-spec } A T$
 by (simp add: RR1-spec-def \mathcal{L} -trim)

lemma *trim-RR2-spec* [simp]:
 $RR2\text{-spec } (trim\text{-reg } A) T \longleftrightarrow RR2\text{-spec } A T$
 by (simp add: RR2-spec-def \mathcal{L} -trim)

lemma *trim-RRn-spec* [simp]:
 $RRn\text{-spec } n (trim\text{-reg } A) T \longleftrightarrow RRn\text{-spec } n A T$
 by (simp add: RRn-spec-def \mathcal{L} -trim)

lemma *swap-RR2-spec*:
 assumes $RR2\text{-spec } A R$
 shows $RR2\text{-spec } (fmap\text{-funs-reg } prod.swap A) (prod.swap ' R)$ 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*:
 $RRn\text{-spec } n empty\text{-reg } \{\}$
 by (auto simp: RRn-spec-def \mathcal{L} -empty)

lemma *true-RR0-spec*:
 $RRn\text{-spec } 0 (Reg \{|q|\} (TA \{\|\ \ \ \ \rightarrow q|\} \{\|\ \ \ \})) \{\ \ \}$
 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* $s \equiv (Some\ s, None)$

abbreviation *lift-None-Some* $s \equiv (None, Some\ s)$

abbreviation $pair\text{-}eps\ A\ B \equiv (\lambda\ (p,\ q).\ ((Some\ (fst\ p),\ q),\ (Some\ (snd\ p),\ q)))\ |\uparrow\ (eps\ A\ |\times|\ finsert\ None\ (Some\ |\uparrow\ \mathcal{Q}\ B))$

abbreviation $pair\text{-}rule \equiv (\lambda\ (ra,\ rb).\ TA\text{-}rule\ (Some\ (r\text{-}root\ ra),\ Some\ (r\text{-}root\ rb))\ (zip\text{-}fill\ (r\text{-}lhs\text{-}states\ ra)\ (r\text{-}lhs\text{-}states\ rb))\ (Some\ (r\text{-}rhs\ ra),\ Some\ (r\text{-}rhs\ rb)))$

lemma $lift\text{-}Some\text{-}None\text{-}prod\text{-}swap\ [simp]:$

$prod.swap \circ lift\text{-}Some\text{-}None = lift\text{-}None\text{-}Some$

$prod.swap \circ lift\text{-}None\text{-}Some = lift\text{-}Some\text{-}None$

by $auto$

lemma $eps\text{-}to\text{-}pair\text{-}eps\text{-}Some\text{-}None:$

$(p,\ q)\ |\in|\ eps\ \mathcal{A} \implies (lift\text{-}Some\text{-}None\ p,\ lift\text{-}Some\text{-}None\ q)\ |\in|\ pair\text{-}eps\ \mathcal{A}\ \mathcal{B}$

by $force$

definition $pair\text{-}automaton :: ('p,\ 'f)\ ta \Rightarrow ('q,\ 'g)\ ta \Rightarrow ('p\ option \times 'q\ option,\ 'f\ option \times 'g\ option)\ ta$ **where**

$pair\text{-}automaton\ A\ B = TA$

$(map\text{-}ta\text{-}rule\ lift\text{-}Some\text{-}None\ lift\text{-}Some\text{-}None\ |\uparrow\ rules\ A\ |\cup|$

$map\text{-}ta\text{-}rule\ lift\text{-}None\text{-}Some\ lift\text{-}None\text{-}Some\ |\uparrow\ rules\ B\ |\cup|$

$pair\text{-}rule\ |\uparrow\ (rules\ A\ |\times|\ rules\ B))$

$(pair\text{-}eps\ A\ B\ |\cup|\ map\text{-}both\ prod.swap\ |\uparrow\ (pair\text{-}eps\ B\ A))$

definition $pair\text{-}automaton\text{-}reg$ **where**

$pair\text{-}automaton\text{-}reg\ R\ L = Reg\ (Some\ |\uparrow\ fin\ R\ |\times|\ Some\ |\uparrow\ fin\ L)\ (pair\text{-}automaton\ (ta\ R)\ (ta\ L))$

lemma $pair\text{-}automaton\text{-}eps\text{-}simps:$

$(lift\text{-}Some\text{-}None\ p,\ p')\ |\in|\ eps\ (pair\text{-}automaton\ A\ B) \longleftrightarrow (lift\text{-}Some\text{-}None\ p,\ p')\ |\in|\ pair\text{-}eps\ A\ B$

$(q,\ lift\text{-}Some\text{-}None\ q')\ |\in|\ eps\ (pair\text{-}automaton\ A\ B) \longleftrightarrow (q,\ lift\text{-}Some\text{-}None\ q')\ |\in|\ pair\text{-}eps\ A\ B$

by $(auto\ simp: pair\text{-}automaton\text{-}def\ eps\text{-}to\text{-}pair\text{-}eps\text{-}Some\text{-}None)$

lemma $pair\text{-}automaton\text{-}eps\text{-}Some\text{-}SomeD:$

$((Some\ p,\ Some\ p'),\ r)\ |\in|\ eps\ (pair\text{-}automaton\ A\ B) \implies fst\ r \neq None \wedge snd\ r \neq None \wedge (Some\ p = fst\ r \vee Some\ p' = snd\ r) \wedge$

$(Some\ p \neq fst\ r \longrightarrow (p,\ the\ (fst\ r))\ |\in|\ (eps\ A)) \wedge (Some\ p' \neq snd\ r \longrightarrow (p',\ the\ (snd\ r))\ |\in|\ (eps\ B))$

by $(auto\ simp: pair\text{-}automaton\text{-}def)$

lemma $pair\text{-}automaton\text{-}eps\text{-}Some\text{-}SomeD2:$

$(r,\ (Some\ p,\ Some\ p'))\ |\in|\ eps\ (pair\text{-}automaton\ A\ B) \implies fst\ r \neq None \wedge snd\ r \neq None \wedge (fst\ r = Some\ p \vee snd\ r = Some\ p') \wedge$

$(fst\ r \neq Some\ p \longrightarrow (the\ (fst\ r),\ p)\ |\in|\ (eps\ A)) \wedge (snd\ r \neq Some\ p' \longrightarrow (the\ (snd\ r),\ p')\ |\in|\ (eps\ B))$

by $(auto\ simp: pair\text{-}automaton\text{-}def)$

lemma $pair\text{-}eps\text{-}Some\text{-}None:$

```

fixes p q q'
defines l  $\equiv$  (p, q) and r  $\equiv$  lift-Some-None q'
assumes (l, r) | $\in$ | (eps (pair-automaton A B))|+|
shows q = None  $\wedge$  p  $\neq$  None  $\wedge$  (the p, q') | $\in$ | (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  $\equiv$  (Some p, Some q)
assumes (l, r) | $\in$ | (eps (pair-automaton A B))|+|
shows fst r  $\neq$  None  $\wedge$  snd r  $\neq$  None  $\wedge$ 
  (fst l  $\neq$  fst r  $\longrightarrow$  (p, the (fst r)) | $\in$ | (eps A)|+|)  $\wedge$ 
  (snd l  $\neq$  snd r  $\longrightarrow$  (q, the (snd r)) | $\in$ | (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  $\equiv$  (Some p, Some q)
assumes (l, r) | $\in$ | (eps (pair-automaton A B))|+|
shows fst l  $\neq$  None  $\wedge$  snd l  $\neq$  None  $\wedge$ 
  (fst l  $\neq$  fst r  $\longrightarrow$  (the (fst l), p) | $\in$ | (eps A)|+|)  $\wedge$ 
  (snd l  $\neq$  snd r  $\longrightarrow$  (the (snd l), q) | $\in$ | (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 ( $\lambda$ (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-union 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-union 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 =  $\lambda X$ . 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  $\subseteq$  eps ?BA rules ?AB  $\subseteq$  rules ?BA
    by (auto simp: fmap-states-ta-def fmap-funs-ta-def pair-automaton-def
      fimage-union 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  $\in$  rules (pair-automaton B A)  $\implies$  ?m r  $\in$  ?m  $\uparrow$  rules (pair-automaton
B A) for r
  by blast
have r  $\in$  rules ?BA  $\implies$  r  $\in$  rules ?AB e  $\in$  eps ?BA  $\implies$  e  $\in$  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  $\subseteq$  eps ?AB rules ?BA  $\subseteq$  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  $\in$  ta-der A (term-of-gterm t)  $\implies$ 
  (Some p, None)  $\in$  ta-der (pair-automaton A B) (term-of-gterm (map-gterm
( $\lambda f$ . (Some f, None)) t))
q  $\in$  ta-der B (term-of-gterm u)  $\implies$ 
  (None, Some q)  $\in$  ta-der (pair-automaton A B) (term-of-gterm (map-gterm
( $\lambda f$ . (None, Some f)) u))
p  $\in$  ta-der A (term-of-gterm t)  $\implies$  q  $\in$  ta-der B (term-of-gterm u)  $\implies$ 
  (Some p, Some q)  $\in$  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)  $\in$  ta-der (pair-automaton A B) (term-of-gterm (map-gterm
( $\lambda f$ . (Some f, None)) s))
  if p  $\in$  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:  $\exists u. q \in | ta\text{-}der\ B\ (term\text{-}of\text{-}gterm\ u) \wedge s = map\text{-}gterm\ (\lambda g. (None, Some\ g))\ u$ 
  if  $(None, Some\ q) \in | ta\text{-}der\ ?AB\ (term\text{-}of\text{-}gterm\ s)$  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:  $h\ rs \rightarrow (sp, sq) \in | rules\ ?AB$ 
     $sp = Some\ p \wedge sq = Some\ q \vee ((sp, sq), (Some\ p, Some\ q)) \in | (eps\ ?AB)|^+$ 
  length rs = length ss
   $\wedge i. i < length\ ss \implies rs\ !\ i \in | ta\text{-}der\ ?AB\ (term\text{-}of\text{-}gterm\ (ss\ !\ i))$ 
  using GFun(2) by auto
  from r(2) have  $sp \neq None\ sq \neq None$  using pair-eps-Some-Some2[of (sp, sq) p q]
  by auto
  then obtain p' q' where pq:  $sp = Some\ p'\ sq = Some\ q'$ 
     $p' = p \vee (p', p) \in | (eps\ A)|^+\ q' = q \vee (q', q) \in | (eps\ B)|^+$ 
  using r(2) pair-eps-Some-Some2[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:  $h = (Some\ f, Some\ g)\ rs = zip\text{-}fill\ ps\ qs$ 
     $f\ ps \rightarrow p' \in | rules\ A\ g\ qs \rightarrow q' \in | rules\ B$ 
  using r(1) unfolding pq(1,2) by (auto simp: pair-automaton-def map-ta-rule-cases)
  have *:  $\exists t\ u. ps\ !\ i \in | ta\text{-}der\ A\ (term\text{-}of\text{-}gterm\ t) \wedge qs\ !\ i \in | ta\text{-}der\ B$ 
     $(term\text{-}of\text{-}gterm\ u) \wedge ss\ !\ i = gpair\ t\ u$ 
  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  $\exists t\ u. (i < length\ ps \implies ps\ !\ i \in | ta\text{-}der\ A\ (term\text{-}of\text{-}gterm\ t)) \wedge$ 
     $(i < length\ qs \implies qs\ !\ i \in | ta\text{-}der\ B\ (term\text{-}of\text{-}gterm\ u)) \wedge$ 
     $ss\ !\ i = gpair\text{-}impl\ (if\ i < length\ ps\ then\ Some\ t\ else\ None)\ (if\ i < length\ qs\ then\ Some\ u\ else\ None)$ 

```

```

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  $\bigwedge i. i < \text{length } ss \implies$ 
      (i < length ps  $\implies$  ps ! i | $\in$ | ta-der A (term-of-gterm (ts i)))  $\wedge$ 
      (i < length qs  $\implies$  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  $\bigwedge i. i < \text{length } ps \implies$  ps ! i | $\in$ | ta-der A (term-of-gterm
(ts i))
       $\bigwedge i. i < \text{length } qs \implies$  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.. $\text{length } ps$ ]]) exI [of - GFun g (map us
[0.. $\text{length } qs$ ])])
      (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 ( $\lambda f. (\text{Some } f, \text{Some } f)$ ) 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  $\mathcal{L}$ -def)
} moreover
{ fix s assume s  $\in$   $\mathcal{L}$  (pair-automaton-reg A B)
      from this [unfolded gta-lang-def  $\mathcal{L}$ -def]

```

```

obtain  $r$  where  $r: r \in \text{fin } (\text{pair-automaton-reg } A \ B) \ r \in \text{gta-der } ?AB \ s$ 
  by (auto simp: pair-automaton-reg-def)
obtain  $p \ q$  where  $pq: r = (\text{Some } p, \text{Some } q) \ p \in \text{fin } A \ q \in \text{fin } B$ 
  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  $p \in \text{ta-der } (ta \ A) \ (\text{term-of-gterm } t) \ q \in \text{ta-der } (ta \ B)$ 
(term-of-gterm  $u$ )  $s = \text{gpair } t \ u$ 
  by (elim exE conjE) note  $*$  = this
then have  $t \in \mathcal{L} \ A \ u \in \mathcal{L} \ B$  using  $pq(2,3)$ 
  by (auto simp: L-def)
then have  $\exists t \ u. s = \text{gpair } t \ u \wedge t \in T \wedge u \in U$  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  $\mathcal{L} \ (\text{pair-automaton-reg } A \ B) = \text{case-prod } \text{gpair} \ ' \ (\mathcal{L} \ A \times \mathcal{L} \ B)$ 
  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  $Qn \ Qs \ (q \ # \ qs) = (\text{let } rec = \text{collapse-state-list } Qn \ Qs \ qs \ \text{in}$ 
  (if  $q \in Qn \wedge q \in Qs$  then map (Cons None) rec @ map (Cons (Some q)) rec
  else if  $q \in Qn$  then map (Cons None) rec
  else if  $q \in Qs$  then map (Cons (Some q)) rec
  else  $[]$ )

```

```

lemma collapse-state-list-inner-length:
  assumes  $qss = \text{collapse-state-list } Qn \ Qs \ qs$ 
  and  $\forall i < \text{length } qs. qs ! i \in Qn \vee qs ! i \in Qs$ 
  and  $i < \text{length } qss$ 
  shows  $\text{length } (qss ! i) = \text{length } qs$  using assms
proof (induct qs arbitrary: qss i)
  case (Cons q qs)
  have  $\forall i < \text{length } qs. qs ! i \in Qn \vee qs ! i \in Qs$  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  $\forall i < \text{length } qs'. qs ! i \in Qn \wedge qs' ! i = \text{None} \vee$ 
   $qs ! i \in Qs \wedge qs' ! i = \text{Some } (qs ! i)$ 
  and  $\text{length } qs = \text{length } qs'$ 
  shows  $qs' \in \text{fset-of-list } (\text{collapse-state-list } Qn \ Qs \ qs)$  using assms
proof (induct qs arbitrary: qs')
  case (Cons q qs)

```

have $(tl\ qs^\wedge) \mid \in \mid$ *fset-of-list* (*collapse-state-list* $Qn\ Qs\ qs$) **using** *Cons*(2-)
by (*intro* *Cons*(1)[*of* $tl\ qs^\wedge$]) (*cases* qs' , *auto*)
then show *?case* **using** *Cons*(2-)
by (*cases* qs^\wedge) (*auto simp: Let-def image-def*)
qed *auto*

lemma *collapse-fset-inv-constr2*:

assumes $\forall i < length\ qs.\ qs\ !\ i \mid \in \mid Qn \vee qs\ !\ i \mid \in \mid Qs$
and $qs' \mid \in \mid$ *fset-of-list* (*collapse-state-list* $Qn\ Qs\ qs$) **and** $i < length\ qs'$
shows $qs\ !\ i \mid \in \mid Qn \wedge qs' \ !\ i = None \vee qs\ !\ i \mid \in \mid Qs \wedge qs' \ !\ i = Some\ (qs\ !\ i)$
using *assms*
proof (*induct* *qs* *arbitrary: qs' i*)
case (*Cons* $a\ qs$)
have $i \neq 0 \implies qs\ !\ (i - 1) \mid \in \mid Qn \wedge tl\ qs' \ !\ (i - 1) = None \vee qs\ !\ (i - 1) \mid \in \mid$
 $Qs \wedge tl\ qs' \ !\ (i - 1) = Some\ (qs\ !\ (i - 1))$
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 $A\ Qn\ Qs =$
 $\mid \cup \mid ((\lambda r.\ fset-of-list\ (map\ (\lambda qs.\ TA-rule\ (r-root\ r)\ qs\ (Some\ (r-rhs\ r))))$
 $(collapse-state-list\ Qn\ Qs\ (r-lhs-states\ r)))) \mid \uparrow$
 $\mid ffilter\ (\lambda r.\ (\forall i < length\ (r-lhs-states\ r).\ r-lhs-states\ r\ !\ i \mid \in \mid Qn \vee r-lhs-states$
 $r\ !\ i \mid \in \mid Qs))$
 $(ffilter\ (\lambda r.\ r-root\ r \neq None)\ (rules\ A))$

definition *collapse-rule-fset where*

collapse-rule-fset $A\ Qn\ Qs = (\lambda r.\ TA-rule\ (the\ (r-root\ r))\ (map\ the\ (filter\ (\lambda q.$
 $\neg Option.is-none\ q)\ (r-lhs-states\ r))))\ (the\ (r-rhs\ r))) \mid \uparrow$
collapse-rule $A\ Qn\ Qs$

lemma *collapse-rule-set-conv*:

fset (*collapse-rule-fset* $A\ Qn\ Qs$) = $\{TA-rule\ f\ (map\ the\ (filter\ (\lambda q.\ \neg Op-$
 $tion.is-none\ q)\ qs'))\ q \mid f\ qs\ qs'\ q.$
 $TA-rule\ (Some\ f)\ qs\ q \mid \in \mid rules\ A \wedge length\ qs = length\ qs' \wedge$
 $(\forall i < length\ qs.\ qs\ !\ i \mid \in \mid Qn \wedge qs' \ !\ i = None \vee qs\ !\ i \mid \in \mid Qs \wedge (qs' \ !\ i) =$
 $Some\ (qs\ !\ i))\}$ (**is** *?Ls* = *?Rs*)

proof

{fix $f\ qs' q\ qs$ **assume** *ass: TA-rule* (*Some* f) $qs\ q \mid \in \mid rules\ A$
 $length\ qs = length\ qs'$
 $\forall i < length\ qs.\ qs\ !\ i \mid \in \mid Qn \wedge qs' \ !\ i = None \vee qs\ !\ i \mid \in \mid Qs \wedge (qs' \ !\ i) =$
 $Some\ (qs\ !\ i)$
then have *TA-rule* (*Some* f) qs' (*Some* q) $\mid \in \mid collapse-rule\ A\ Qn\ Qs$
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 ($\lambda q. \neg \text{Option.is-none } q$) qs')) $q \in ?Ls$
unfolding *collapse-rule-fset-def*
by (auto simp: image-iff Bex-def intro!: exI[of - *TA-rule* (Some f) qs' (Some q)])
then show $?Rs \subseteq ?Ls$ **by** blast
next
{fix f qs q **assume** *ass*: *TA-rule* f qs $q \in ?Ls$
then obtain ps qs' **where** w : *TA-rule* (Some f) ps $q \in$ rules A
 $\forall i < \text{length } ps. ps ! i \in Qn \vee ps ! i \in Qs$
 $qs' \in \text{fset-of-list } (\text{collapse-state-list } Qn \ Qs \ ps)$
 $qs = \text{map the } (\text{filter } (\lambda q. \neg \text{Option.is-none } q) \ qs')$
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 $\forall i < \text{length } qs'. ps ! i \in Qn \wedge qs' ! i = \text{None} \vee ps ! i \in Qs \wedge$
 $qs' ! i = \text{Some } (ps ! i)$
using *collapse-fset-inv-constr2*
by metis
moreover have $\text{length } ps = \text{length } qs'$
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 $q \in ?Rs$
using $w(1)$ **unfolding** $w(4)$
by auto fastforce}
then show $?Ls \subseteq ?Rs$
by (intro subsetI) (metis (no-types, lifting) *ta-rule.collapse*)
qed

lemma *collapse-rule-fmember* [simp]:

TA-rule f qs $q \in$ (collapse-rule-fset $A \ Qn \ Qs$) $\longleftrightarrow (\exists qs' ps.$
 $qs = \text{map the } (\text{filter } (\lambda q. \neg \text{Option.is-none } q) \ qs') \wedge \text{TA-rule } (Some \ f) \ ps \ q \in$
rules $A \wedge \text{length } ps = \text{length } qs' \wedge$
 $(\forall i < \text{length } ps. ps ! i \in Qn \wedge qs' ! i = \text{None} \vee ps ! i \in Qs \wedge (qs' ! i) = \text{Some}$
 $(ps ! i)))$
unfolding *collapse-rule-set-conv*
by auto

definition $Qn \ A \equiv (\text{let } S = (\text{r-rhs } |' \ \text{ffilter } (\lambda r. \text{r-root } r = \text{None}) \ (\text{rules } A)) \ \text{in}$
 $(\text{eps } A)^+ |' \ S \ |\cup| \ S)$

definition $Qs \ A \equiv (\text{let } S = (\text{r-rhs } |' \ \text{ffilter } (\lambda r. \text{r-root } r \neq \text{None}) \ (\text{rules } A)) \ \text{in}$
 $(\text{eps } A)^+ |' \ S \ |\cup| \ S)$

lemma *Qn-member-iff* [simp]:

$q \in Qn \ A \longleftrightarrow (\exists ps \ p. \text{TA-rule } \text{None} \ ps \ p \in \text{rules } A \wedge (p = q \vee (p, q) \in$
 $(\text{eps } A)^+))$ (is $?Ls \longleftrightarrow ?Rs$)

proof –

{assume *ass*: $q \in Qn \ A$ **then obtain** r **where**
 $\text{r-rhs } r = q \vee (\text{r-rhs } r, q) \in (\text{eps } A)^+ \mid r \in \text{rules } A \ \text{r-root } r = \text{None}$

by (force simp: Qn-def Image-def image-def Let-def fImage.rep-eq)
 then have ?Ls \implies ?Rs by (cases r) auto
 moreover have ?Rs \implies ?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]:

$q \in | Qs A \iff (\exists f ps p. TA\text{-rule } (Some f) ps p \in | rules A \wedge (p = q \vee (p, q) \in | (eps A)^+))$ (is ?Ls \iff ?Rs)

proof –

{assume ass: $q \in | Qs A$ then obtain $f r$ where
 $r\text{-rhs } r = q \vee (r\text{-rhs } r, q) \in | (eps A)^+ | r \in | rules A$ $r\text{-root } r = Some f$
 by (force simp: Qs-def Image-def image-def Let-def fImage.rep-eq)
 then have ?Ls \implies ?Rs by (cases r) auto
 moreover have ?Rs \implies ?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:

$fset (Qn A) = \{q' \mid qs q q'. TA\text{-rule } None qs q \in | rules A \wedge (q = q' \vee (q, q') \in | (eps A)^+)|\}$ (is ?Ls1 = ?Rs1)
 $fset (Qs A) = \{q' \mid f qs q q'. TA\text{-rule } (Some f) qs q \in | rules A \wedge (q = q' \vee (q, q') \in | (eps A)^+)|\}$ (is ?Ls2 = ?Rs2)
 by auto force+

definition collapse-automaton :: ('q, 'f option) ta \Rightarrow ('q, 'f) ta **where**
 collapse-automaton A = TA (collapse-rule-fset A (Qn A) (Qs A)) (eps A)

definition collapse-automaton-reg **where**

collapse-automaton-reg R = Reg (fin R) (collapse-automaton (ta R))

lemma ta-states-collapse-automaton:

Q (collapse-automaton A) \subseteq | 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  $\mathcal{Q} A \sqsubseteq | ta\text{-reachable } A$ 
shows  $gta\text{-lang } Q \text{ (collapse-automaton } A) = \text{the } \langle gcollapse \text{ } \langle gta\text{-lang } Q A - \{None\} \rangle \rangle$ 
proof -
  define  $Q_{n'}$  where  $Q_{n'} = Q_n A$ 
  define  $Q_{s'}$  where  $Q_{s'} = Q_s A$ 
  {fix  $t$  assume  $t: t \in gta\text{-lang } Q \text{ (collapse-automaton } A)$ 
  then obtain  $q$  where  $q: q \in | Q q \in | ta\text{-der (collapse-automaton } A) \text{ (term-of-gterm$ 
 $t)$  by auto
    have  $\exists t'. q \in | ta\text{-der } A \text{ (term-of-gterm } t') \wedge gcollapse \ t' \neq None \wedge t = \text{the}$ 
 $(gcollapse \ t')$  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  $ps: TA\text{-rule (Some } f) \ qs \ p \in | \text{rules}$ 
 $A \text{ length } qs = \text{length } rs$ 
           $\wedge i. i < \text{length } qs \implies qs \ ! \ i \in | Q_{n'} \wedge rs \ ! \ i = None \vee qs \ ! \ i \in | Q_{s'} \wedge rs \ !$ 
 $i = \text{Some } (qs \ ! \ i)$ 
           $ps = \text{map the (filter } (\lambda q. \neg Option.is\text{-none } q) \ rs)$ 
          by (auto simp: collapse-automaton-def Qn'-def Qs'-def)
          obtain  $ts'$  where  $ts'$ :
             $ps \ ! \ i \in | ta\text{-der } A \text{ (term-of-gterm (ts' } i)) \ gcollapse \ (ts' \ i) \neq None \ ts \ ! \ i =$ 
 $\text{the } (gcollapse \ (ts' \ i))$ 
            if  $i < \text{length } ts$  for  $i$  using  $GFun(5)$  by metis
            from  $ps(2, 3, 4)$  have  $rs: i < \text{length } qs \implies rs \ ! \ i = None \vee rs \ ! \ i = \text{Some}$ 
 $(qs \ ! \ i)$  for  $i$ 
            by auto
            {fix  $i$  assume  $i < \text{length } qs \ rs \ ! \ i = None$ 
            then have  $\exists t'. \text{groot-sym } t' = None \wedge qs \ ! \ i \in | ta\text{-der } A \text{ (term-of-gterm}$ 
 $t')$ 
              using  $ps(1, 2) \ ps(3)[\text{of } i]$ 
              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  $None = \text{this}$ 
            {fix  $i$  assume  $i: i < \text{length } qs \ rs \ ! \ i = \text{Some } (qs \ ! \ i)$ 
            have  $\text{map } \text{Some } ps = \text{filter } (\lambda q. \neg Option.is\text{-none } q) \ rs$  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  $*$ :  $rs \ ! \ i = \text{map } \text{Some } ps \ ! \ \text{filter-rev-nth } (\lambda q. \neg Option.is\text{-none } q) \ rs \ i$ 
 $\text{filter-rev-nth } (\lambda q. \neg Option.is\text{-none } q) \ rs \ i < \text{length } ps$ 
            using  $ps(2, 4) \ i$  by auto
            then have  $\text{the } (rs \ ! \ i) = ps \ ! \ \text{filter-rev-nth } (\lambda q. \neg Option.is\text{-none } q) \ rs \ i$ 
 $\text{filter-rev-nth } (\lambda q. \neg Option.is\text{-none } q) \ rs \ i < \text{length } ps$ 
            by auto } note  $\text{Some} = \text{this}$ 
            let  $?P = \lambda t \ i. qs \ ! \ i \in | ta\text{-der } A \text{ (term-of-gterm } t) \wedge$ 
 $(rs \ ! \ i = None \implies \text{groot-sym } t = None) \wedge$ 
 $(rs \ ! \ i = \text{Some } (qs \ ! \ i) \implies t = ts' \ (\text{filter-rev-nth } (\lambda q. \neg Option.is\text{-none } q)$ 
 $rs \ i))$ 
            {fix  $i$  assume  $i: i < \text{length } qs$ 

```

```

then have  $\exists t. ?P t i$  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  $\implies$  qs ! i | $\in$ | ta-der A (term-of-gterm (ts'' ! i))
  i < length qs  $\implies$  rs ! i = None  $\implies$  groot-sym (ts'' ! i) = None
  i < length qs  $\implies$  rs ! i = Some (qs ! i)  $\implies$  ts'' ! i = ts' (filter-rev-nth ( $\lambda q.$ 
 $\neg$  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 ( $\lambda q.$   $\neg$  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)  $\implies$  butlast rs ! i = None  $\implies$  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)  $\implies$  butlast rs ! i = None  $\vee$  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)  $\implies$  butlast rs ! i = Some (butlast qs ! i)  $\implies$ 
us ! i = ts' (filter-rev-nth ( $\lambda q.$   $\neg$  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 ( $\lambda q.$   $\neg$  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 ( $\lambda q.$   $\neg$  Option.is-none q) rs (length us)
                      have map Some (butlast ts) = filter ( $\lambda q.$   $\neg$  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  $a = ts' ?i ?i < \text{length } ps$ 
  using  $\text{False snoc}(2, 9) \text{ l snoc}(4, 6, 10)[\text{of length } us]$ 
  by  $(\text{auto simp: nth-append})$ 
moreover have  $\text{filter-rev-nth } (\lambda q. \neg \text{Option.is-none } q) (rs' @ [r]) (\text{length } rs') = \text{length } ss$ 
  using  $\text{l snoc}(2, 7, 8, 9) \text{ False unfolding } *$ 
  by  $(\text{auto simp: filter-rev-nth-def})$ 
ultimately show  $?thesis$  using  $\text{l snoc}(2, 7, 9) \text{ snoc}(11-)[\text{of } ?i]$ 
  by  $(\text{auto simp: nth-append } *)$ 
qed
qed simp
then have  $ts = \text{map the } (\text{filter } (\lambda t. \neg \text{Option.is-none } t) (\text{map } gcollapse \text{ ts}'))$ 
  by  $(\text{metis comp-the-Some list.map-id map-map})$ 
then show  $?case$  using  $ps(1, 2) \text{ ts}''(1, 2) \text{ GFun}(3)$ 
  by  $(\text{auto simp: collapse-automaton-def intro!: exI}[\text{of - GFun (Some f) ts}''])$ 
exI}[\text{of - qs}] \text{ exI}[\text{of - p}]
qed
then have  $t \in \text{the } (gcollapse \text{ ' gta-lang } Q \ A - \{None\})$ 
  by  $(\text{meson Diff-iff gta-langI imageI } q(1) \text{ singletonD})$ 
} moreover
{ fix } t \text{ assume } t: t \in \text{gta-lang } Q \ A \text{ gcollapse } t \neq \text{None}
  obtain } q \text{ where } q: q \in Q \ q \in \text{ta-der } A \text{ (term-of-gterm } t) \text{ using } t(1) \text{ by}
 $\text{auto}$ 
  have } q \in \text{ta-der } (collapse-automaton \ A) \text{ (term-of-gterm (the (gcollapse } t))
using } q(2) \ t(2)
  proof  $(\text{induct } t \text{ arbitrary: } q)$ 
  case  $(\text{GFun } f \ ts)$ 
  obtain } qs \ q' \text{ where } q: \text{TA-rule } f \ qs \ q' \in \text{rules } A \ q' = q \vee (q', q) \in \text{eps}
 $(collapse-automaton \ A)|^+$ 
   $\text{length } qs = \text{length } ts \wedge i. i < \text{length } ts \implies qs ! i \in \text{ta-der } A \text{ (term-of-gterm}$ 
 $(ts ! i))$ 
  using  $\text{GFun}(2) \text{ by } (\text{auto simp: collapse-automaton-def})$ 
  obtain } f' \text{ where } f \text{ [simp]: } f = \text{Some } f' \text{ using } \text{GFun}(3) \text{ by } (\text{cases } f) \text{ auto}
  define } qs' \text{ where}
   $qs' = \text{map } (\lambda i. \text{if } \text{Option.is-none } (gcollapse \ (ts ! i)) \text{ then } \text{None} \text{ else } \text{Some}$ 
 $(qs ! i)) [0..<\text{length } qs]$ 
  have  $\text{Option.is-none } (gcollapse \ (ts ! i)) \implies qs ! i \in Qn' \text{ if } i < \text{length } qs \text{ for}$ 
 $i$ 
  using  $q(4)[\text{of } i] \text{ that}$ 
  by  $(\text{cases } ts ! i \text{ rule: } gcollapse.cases)$ 
   $(\text{auto simp: } q(3) \ Qn'\text{-def } collapse-Qn-Qs\text{-set-conv})$ 
  moreover have  $\neg \text{Option.is-none } (gcollapse \ (ts ! i)) \implies qs ! i \in Qs' \text{ if } i$ 
 $< \text{length } qs \text{ for } i$ 
  using  $q(4)[\text{of } i] \text{ that}$ 
  by  $(\text{cases } ts ! i \text{ rule: } gcollapse.cases)$ 
   $(\text{auto simp: } q(3) \ Qs'\text{-def } collapse-Qn-Qs\text{-set-conv})$ 
  ultimately have  $f' (\text{map the } (\text{filter } (\lambda q. \neg \text{Option.is-none } q) \ qs')) \rightarrow q' \in$ 
 $\text{rules } (collapse-automaton \ A)$ 
  using  $q(1, 4) \text{ unfolding } collapse-automaton-def \ Qn'\text{-def}[\text{symmetric}] \ Qs'\text{-def}[\text{symmetric}]$ 

```

```

    by (fastforce simp: qs'-def q(3) intro: exI[of - qs] exI[of - qs'] split: if-splits)
  moreover have ***: length (filter (λi. ¬ Option.is-none (gcollapse (ts ! i)))
[0..

```

```

      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 \mathcal{L} -collapse-automaton':
 assumes $\mathcal{Q}_r A \mid\subseteq\mid$ ta-reachable (ta A)
 shows \mathcal{L} (collapse-automaton-reg A) = the ' (gcollapse ' \mathcal{L} A - {None})
 using assms by (auto simp: collapse-automaton-reg-def \mathcal{L} -def collapse-automaton')

lemma collapse-automaton:
 assumes $\mathcal{Q}_r A \mid\subseteq\mid$ ta-reachable (ta A) RR1-spec A T
 shows RR1-spec (collapse-automaton-reg A) (the ' (gcollapse ' \mathcal{L} A - {None}))
 using collapse-automaton'[OF assms(1)] assms(2)
 by (simp add: collapse-automaton-reg-def \mathcal{L} -def RR1-spec-def)

5.9 Cylindrification

definition pad-with-Nones where

$pad\text{-with-Nones } n m = (\lambda(f, g). case\text{-option } (replicate\ n\ None)\ id\ f\ @\ case\text{-option } (replicate\ m\ None)\ id\ g)$

lemma gencode-append:

$gencode (ss @ ts) = map\text{-gterm } (pad\text{-with-Nones } (length\ ss)\ (length\ ts))\ (gpair\ (gencode\ ss)\ (gencode\ ts))$

proof –

have [simp]: $p \notin gposs (gunions (map\ gdomain\ ts)) \implies map (\lambda t. gfun\text{-at } t\ p)\ ts = replicate (length\ ts)\ None$

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 $\lambda(- :: unit\ option). ()$, unfolded comp-def gdomain-id]

show ?thesis by (auto intro!: arg-cong[of - - $\lambda x. glabel\ x\ -$] 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** $ts \in T \ us \in U \ s = \text{gencode } (ts \ @ \ us)$
by (*fastforce simp: \mathcal{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 \mathcal{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 ($\lambda(f, g). \text{pad-with-Nones } 1 \ m \ (\text{map-option } (\lambda f. [\text{Some } f]) \ f, \ g))$
(pair-automaton-reg A B)) {*t # us* | *t us. t ∈ T ∧ us ∈ U*}
proof –
have [*simp*]: {*ts @ us* | *ts us. ts ∈ ($\lambda t. [t]$) ‘ 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: \mathcal{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* \equiv *if list-all (Option.is-none) (drop m fs) then None else Some (drop m fs)*

lemma *drop-automaton-reg:*

assumes $\mathcal{Q}_r \ A \ |\subseteq| \ ta \ \text{reachable } (ta \ A) \ m < n \ RRn \ \text{spec } n \ A \ T$
defines $f \equiv \lambda fs. \text{drop-none-rule } m \ fs$
shows *RRn-spec (n – m) (collapse-automaton-reg (fmap-funs-reg f A)) (drop m ‘ T)*
proof –
have [*simp*]: $\text{length } ts = n - m \implies p \in \text{gposs } (\text{gencode } ts) \implies \exists f. \exists t \in \text{set } ts. \text{Some } f = \text{gfun-at } t \ p \ \text{for } p \ ts$
proof (*cases p, goal-cases Empty PCons*)
case *Empty*
obtain *t* **where** $t \in \text{set } ts$ **using** *assms(2) Empty(1)* **by** (*cases ts*) *auto*
moreover then obtain *f* **where** $\text{Some } f = \text{gfun-at } t \ p$ **using** *Empty(3)* **by** (*cases t rule: gterm.exhaust*) *auto*
ultimately show ?*thesis* **by** *auto*
next
case (*PCons i p'*)
then have $p \neq []$ **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: 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 A) = *the* ‘ (gcollapse ‘ map-gterm fst ‘ (\mathcal{L} A) – {None})

proof –

have \mathcal{Q}_r (fmap-funs-reg fst (trim-reg A)) \sqsubseteq | *ta-reachable* (ta (fmap-funs-reg fst (trim-reg 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 A) = *the* ‘ (gcollapse ‘ map-gterm snd ‘ (\mathcal{L} A) – {None})

proof –

have \mathcal{Q}_r (fmap-funs-reg snd (trim-reg A)) \sqsubseteq | *ta-reachable* (ta (fmap-funs-reg snd (trim-reg 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 = *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 ∈ \mathcal{L} (proj-1-reg A) **using** *assms* *ass* s

by (auto simp: proj-1-reg-simp \mathcal{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 $s \in \mathcal{L}$ (proj-1-reg A) then have $s \in \text{fst} \text{ ' } R$ 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 \text{ ' } R)*
proof –
{fix s t assume *ass*: $(s, t) \in R$
then have $s: t = \text{the } (gcollapse (map-gterm snd (gpair s t)))$
by (auto simp: *gsnd-gpair gsnd-collapse-simp*)
then have $\text{Some } t = gcollapse (map-gterm snd (gpair s t))$
by (cases s; cases t) (auto simp: *gpair-def*)
then have $t \in \mathcal{L}$ (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 $s \in \mathcal{L}$ (proj-2-reg A) then have $s \in \text{snd} \text{ ' } R$ 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 \mathcal{L} (proj-1-reg A) = *gfst \text{ ' } \mathcal{L} A \mathcal{L} (proj-2-reg A) = *gsnd \text{ ' } \mathcal{L} A**
proof –
have [*simp*]: *gfst \text{ ' } \{gpair t u | t u. (t, u) \in R\} = fst \text{ ' } R*
by (force simp: *gfst-gpair image-def*)
have [*simp*]: *gsnd \text{ ' } \{gpair t u | t u. (t, u) \in R\} = snd \text{ ' } R*
by (force simp: *gsnd-gpair image-def*)
show \mathcal{L} (proj-1-reg A) = *gfst \text{ ' } \mathcal{L} A \mathcal{L} (proj-2-reg A) = *gsnd \text{ ' } \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* ($\lambda xs.$ *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* ($\lambda xs.$ *map* (!) *xs ps*) *A*) (($\lambda xs.$ *map* (!) *xs ps*) ‘ *T*)

using *assms* by (*auto simp: RRn-spec-def gencode-permute fmap-funs-reg-def 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* \cap *U*) using *assms*

by (*simp add: RRn-spec-def 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* \cup *U*)

using *assms* by (*auto simp: RRn-spec-def 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 L-difference-reg*)

lemma *RR2-difference*:

assumes *RR2-spec A T RR2-spec B U*

shows $RR2\text{-spec}$ ($\text{difference-reg } A B$) ($T - U$)
using *assms* **by** (*auto simp: RR2-spec-def \mathcal{L} -difference-reg*)

lemma $RRn\text{-difference}$:

assumes $RRn\text{-spec } n A T RRn\text{-spec } n B U$
shows $RRn\text{-spec } n$ ($\text{difference-reg } A B$) ($T - U$)
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 :: $(f \times \text{nat}) \text{fset} \Rightarrow (\text{unit}, f) \text{ta}$ **where**
 $\text{term-automaton } \mathcal{F} = \text{TA } ((\lambda (f, n). \text{TA-rule } f (\text{replicate } n ()) ()) \mid \mathcal{F}) \{\mid\}$

definition term-reg **where**
 $\text{term-reg } \mathcal{F} = \text{Reg } \{\mid()\mid\} (\text{term-automaton } \mathcal{F})$

lemma term-automaton :

$RR1\text{-spec}$ ($\text{term-reg } \mathcal{F}$) (\mathcal{T}_G ($\text{fset } \mathcal{F}$))
unfolding $RR1\text{-spec-def gta-lang-def term-reg-def } \mathcal{L}\text{-def}$

proof (*intro set-eqI iffI, goal-cases lr rl*)

case ($lr t$)

then have $() \mid \in \mid \text{ta-der}$ ($\text{term-automaton } \mathcal{F}$) ($\text{term-of-gterm } t$)

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 $() \mid \in \mid \text{ta-der}$ ($\text{term-automaton } \mathcal{F}$) ($\text{term-of-gterm } t$)

proof (*induct t rule: \mathcal{T}_G .induct*)

case ($\text{const } a$) **then show** $?case$

by (*auto simp: term-automaton-def image-iff intro: bexI[of - (a, 0)]*)

next

case ($\text{ind } f n ss$) **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 :: $(f \times \text{nat}) \text{fset} \Rightarrow \text{nat} \Rightarrow (\text{nat}, f \text{ option list}) \text{reg}$ **where**

$\text{true-RRn } \mathcal{F} 0 = \text{Reg } \{\mid 0 \mid\} (\text{TA } \{\mid \text{TA-rule } [] [] 0 \mid\} \{\mid\})$

$\mid \text{true-RRn } \mathcal{F} (\text{Suc } 0) = \text{relabel-reg } (\text{fmap-funs-reg } (\lambda f. [\text{Some } f]) (\text{term-reg } \mathcal{F}))$

$\mid \text{true-RRn } \mathcal{F} (\text{Suc } n) = \text{relabel-reg}$

$(\text{trim-reg } (\text{fmap-funs-reg } (\text{pad-with-Nones } 1 n) (\text{pair-automaton-reg } (\text{true-RRn } \mathcal{F} 1) (\text{true-RRn } \mathcal{F} n))))$

lemma true-RRn-spec :

$RRn\text{-spec } n$ ($\text{true-RRn } \mathcal{F} n$) $\{ts. \text{length } ts = n \wedge \text{set } ts \subseteq \mathcal{T}_G (\text{fset } \mathcal{F})\}$

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  $\{ts. \text{length } ts = 1 \wedge \text{set } ts \subseteq \mathcal{T}_G (\text{fset } \mathcal{F})\} = (\lambda t. [t]) \cdot \mathcal{T}_G (\text{fset } \mathcal{F})$ 
  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]:  $\{ts @ us \mid ts \text{ us. } \text{length } ts = n \wedge \text{set } ts \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \wedge \text{length } us = m \wedge \text{set } us \subseteq \mathcal{T}_G (\text{fset } \mathcal{F})\} = \{ss. \text{length } ss = n + m \wedge \text{set } ss \subseteq \mathcal{T}_G (\text{fset } \mathcal{F})\}$  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 $RR2\text{-to-}RRn$ $A \equiv \text{fmap-funs-reg } (\lambda(f, g). [f, g]) A$

abbreviation $RRn\text{-to-}RR2$ $A \equiv \text{fmap-funs-reg } (\lambda f. (f ! 0, f ! 1)) A$

definition $rr2\text{-compositon}$ **where**

```

  rr2-compositon  $\mathcal{F}$   $A$   $B$  =
    (let  $A' = RR2\text{-to-}RRn$   $A$  in
     let  $B' = RR2\text{-to-}RRn$   $B$  in
     let  $F = \text{true-}RRn$   $\mathcal{F}$  1 in
     let  $CA = \text{trim-reg } (\text{fmap-funs-reg } (\text{pad-with-Nones } 2 \ 1) (\text{pair-automaton-reg } A' F))$  in
     let  $CB = \text{trim-reg } (\text{fmap-funs-reg } (\text{pad-with-Nones } 1 \ 2) (\text{pair-automaton-reg } F B'))$  in
     let  $PI = \text{trim-reg } (\text{fmap-funs-reg } (\lambda xs. \text{map } (!) xs) [1, 0, 2]) (\text{reg-intersect } CA CB))$  in
      $RRn\text{-to-}RR2$  ( $\text{collapse-automaton-reg } (\text{fmap-funs-reg } (\text{drop-none-rule } 1) PI)$ )
    )

```

lemma list-length1E :

```

  assumes  $\text{length } xs = \text{Suc } 0$  obtains  $x$  where  $xs = [x]$  using  $\text{assms}$ 
  by (cases  $xs$ ) auto

```

lemma $rr2\text{-compositon}$:

```

  assumes  $\mathcal{R} \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \times \mathcal{T}_G (\text{fset } \mathcal{F})$   $\mathcal{L} \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \times \mathcal{T}_G (\text{fset } \mathcal{F})$ 
  and  $RR2\text{-spec } A$   $\mathcal{R}$  and  $RR2\text{-spec } B$   $\mathcal{L}$ 
  shows  $RR2\text{-spec } (rr2\text{-compositon } \mathcal{F} A B)$   $(\mathcal{R} \ O \ \mathcal{L})$ 

```

proof –

```

let ?R = ( $\lambda(t, u). [t, u]$ ) ‘ $\mathcal{R}$  let ?L = ( $\lambda(t, u). [t, u]$ ) ‘ $\mathcal{L}$ 
let ?A = RR2-to-RRn A let ?B = RR2-to-RRn B let ?F = true-RRn 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 ( $\lambda xs. \text{map } (!) xs$ ) [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  $\in$  ?R  $\wedge$  ( $\exists t. us = [t] \wedge t \in \mathcal{T}_G$  (fset F))}
let ?Ls = {us @ ts | ts us. ts  $\in$  ?L  $\wedge$  ( $\exists t. us = [t] \wedge t \in \mathcal{T}_G$  (fset 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  $\wedge$  set ts  $\subseteq \mathcal{T}_G$  (fset F)} = {[t | t. t  $\in \mathcal{T}_G$  (fset F)]}
by (auto elim!: list-length1E)
have F: RRn-spec 1 ?F {[t | t. t  $\in \mathcal{T}_G$  (fset F)]} using true-RRn-spec[of 1 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 (( $\lambda xs. \text{map } (!) xs$ ) [1, 0, 2]) ‘(?Rs  $\cap$  ?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 ‘( $\lambda xs. \text{map } (!) xs$ ) [1, 0, 2]) ‘(?Rs  $\cap$  ?Ls)
by auto
{fix s assume s  $\in$  ( $\lambda(t, u). [t, u]$ ) ‘( $\mathcal{R} \ O \ \mathcal{L}$ )
then obtain t u v where comp: s = [t, u] (t, v)  $\in$   $\mathcal{R}$  (v, u)  $\in$   $\mathcal{L}$ 
by (auto simp: image-iff relcomp-unfold split!: prod.split)
then have [t, v]  $\in$  ?R [v, u]  $\in$  ?L u  $\in \mathcal{T}_G$  (fset F) v  $\in \mathcal{T}_G$  (fset F) t  $\in \mathcal{T}_G$ 
(fset F) using assms(1, 2)
by (auto simp: image-iff relcomp-unfold split!: prod.splits)
then have [t, v, u]  $\in$  ?Rs [t, v, u]  $\in$  ?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  $\in$  drop 1 ‘( $\lambda xs. \text{map } (!) xs$ ) [1, 0, 2]) ‘(?Rs  $\cap$  ?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 $\text{drop } 1 \text{ ' } (\lambda xs. \text{map } (!) \text{ } xs) [1, 0, 2]) \text{ ' } (?Rs \cap ?Ls) \subseteq (\lambda(t, u). [t, u]) \text{ ' } (\mathcal{R} \text{ } O \ \mathfrak{L})$
by (*auto simp: image-iff relcomp-unfold Bex-def split!: prod.splits*)
ultimately have $*$: $\text{drop } 1 \text{ ' } (\lambda xs. \text{map } (!) \text{ } xs) [1, 0, 2]) \text{ ' } (?Rs \cap ?Ls) = (\lambda(t, u). [t, u]) \text{ ' } (\mathcal{R} \text{ } O \ \mathfrak{L})$
by (*simp add: subsetI subset-antisym*)
have $**$: $(\lambda f. (f ! 0, f ! 1)) \text{ ' } (\lambda(t, u). [t, u]) \text{ ' } (\mathcal{R} \text{ } O \ \mathfrak{L}) = \mathcal{R} \text{ } O \ \mathfrak{L}$
by (*force simp: image-def relcomp-unfold split!: prod.splits*)
show *?thesis* **using** *sp unfolding **
using *RRn-to-RR2-spec* [**where** $?T = (\lambda(t, u). [t, u]) \text{ ' } (\mathcal{R} \text{ } O \ \mathfrak{L})$ **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*]: $\text{map-ta-rule } f \text{ id } r = (r\text{-root } r) (\text{map } f (r\text{-lhs-states } r)) \rightarrow (f (r\text{-rhs } r))$ **for** $f \ r$
by (*simp add: ta-rule.expand ta-rule.map-sel(1 - 3)*)

lemma *no-upper-bound-infinite*:
assumes $\forall (n::\text{nat}). \exists t \in S. n < f \ t$
shows *infinite* S
proof (*rule ccontr, simp*)
assume *finite* S
then obtain n **where** $n = \text{Max } (f \text{ ' } S) \ \forall t \in S. f \ t \leq n$ **by** *auto*
then show *False* **using** *assms linorder-not-le* **by** *blast*
qed

lemma *set-constr-finite*:
assumes *finite* F
shows *finite* $\{h \ x \mid x. x \in F \wedge P \ x\}$ **using** *assms*
by (*induct*) *auto*

lemma *bounded-depth-finite*:
assumes *fin-F*: *finite* \mathcal{F} **and** $\bigcup (\text{funas-term ' } S) \subseteq \mathcal{F}$
and $\forall t \in S. \text{depth } t \leq n$ **and** $\forall t \in S. \text{ground } t$
shows *finite* S **using** *assms(2-)*
proof (*induction n arbitrary: S*)
case 0
{fix t **assume** *elem*: $t \in S$
from 0 **have** $\text{depth } t = 0$ **ground** t *funas-term* $t \subseteq \mathcal{F}$ **using** *elem* **by** *auto*
then have $\exists f. (f, 0) \in \mathcal{F} \wedge t = \text{Fun } f \ []$ **by** (*cases t rule: depth.cases*) *auto*}

```

then have  $S \subseteq \{Fun\ f\ []\ |f \cdot (f, 0) \in \mathcal{F}\}$  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
     $S: S' = \{t :: ('a, 'b)\ term \cdot ground\ t \wedge funas-term\ t \subseteq \mathcal{F} \wedge depth\ t \leq n\}$  finite
   $S'$ 
    by (auto simp add: SUP-le-iff)
  then obtain  $L\ F$  where  $L: set\ L = S'$   $F = \mathcal{F}$  using fin-F by (meson
  finite-list)
  let ?Sn =  $\{Fun\ f\ ts\ |f\ ts. (f, length\ ts) \in \mathcal{F} \wedge set\ ts \subseteq S'\}$ 
  let ?Ln = concat (map ( $\lambda (f, n). map\ (\lambda ts. Fun\ f\ ts)\ (List.n-lists\ n\ L)$ ) 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:
 $infinite\ (f\ 'S) \implies inj-on\ f\ S \implies infinite\ S$
by blast

lemma infinite-imageD2:
 $infinite\ (f\ 'S) \implies inj\ f \implies infinite\ S$
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 F*
and $\forall t \in S. \text{funas-term } t \subseteq \mathcal{F}$ **and** $\forall t \in S. \text{ground } t$
shows $\forall (n::\text{nat}). \exists t \in S. n < (\text{depth } t)$
proof(*rule allI, rule ccontr*)
fix $n::\text{nat}$
assume $\neg (\exists t \in S. (\text{depth } t) > n)$
hence $\forall t \in S. \text{depth } t \leq n$ **by** *auto*
from *bounded-depth-finite[OF assms(2) - this]* **show** *False using assms*
by *auto*
qed

lemma *depth-gterm-conv*:
 $\text{depth } (\text{term-of-gterm } t) = \text{depth } (\text{term-of-gterm } t)$
by (*metis leD nat-neq-iff poss-gposs-conv poss-length-bounded-by-depth poss-length-depth*)

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

lemma *pigeonhole-ta-infinit-terms*:
fixes $t :: 'f \text{ gterm}$ **and** $\mathcal{A} :: ('q, 'f) \text{ ta}$
defines $t' \equiv \text{term-of-gterm } t :: ('f, 'q) \text{ term}$
assumes $\text{fcard } (\mathcal{Q} \mathcal{A}) < \text{depth } t'$ **and** $q \in | \text{gta-der } \mathcal{A} \ t$ **and** $P (\text{funas-gterm } t)$
shows *infinite* $\{t . q \in | \text{gta-der } \mathcal{A} \ t \wedge P (\text{funas-gterm } t)\}$
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⟨s⟩ = t' C2⟨v⟩ = s and*
loop: p ∈ | ta-der A v p ∈ | ta-der A C2⟨Var p⟩ q ∈ | ta-der A C⟨Var p⟩
unfolding *assms(1)* **by** *auto*
let $?terms = \lambda n. C\langle (C2 \hat{\ } n)\langle v \rangle \rangle$ **let** $?inner = \lambda n. (C2 \hat{\ } n)\langle v \rangle$
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 A (?terms (Suc n)) for n using loop*
by (*meson ta-der-ctxt ta-der-ctxt-n-loop*)
moreover **have** $n < \text{depth } (?terms (Suc n))$ **for** n
by (*meson ctxt(1) ctxt-comp-n-lower-bound depth-ctxt-less-eq less-le-trans*)
ultimately **have** $q \in | \text{ta-der } \mathcal{A} \ (?terms (Suc n)) \wedge \text{ground } (?terms (Suc n)) \wedge$
 $P (\text{funas-term } (?terms (Suc n))) \wedge n < \text{depth } (?terms (Suc n))$ **for** n **using**
assms(4)
by (*auto simp: assms(1) funas-term-of-gterm-conv*)
then **have** *inf: infinite {t. q ∈ | ta-der 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 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) \subseteq (λ (f, n). ((None, Some f), n)) ‘ \mathcal{F} \longleftrightarrow
funas-gterm t \subseteq \mathcal{F}
by (induct t) (auto simp: funas-gterm-def, blast)

lemma funas-gterm-bot-some-decomp:
assumes funas-gterm s \subseteq (λ (f, n). ((None, Some f), n)) ‘ \mathcal{F}
shows \exists t. gterm-to-None-Some t = s \wedge funas-gterm t \subseteq \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 \wedge (\forall 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 \implies funas-gterm (ss ! i) \subseteq \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} $\mathcal{F} = \{t . \text{infinite } \{u. (t, u) \in \mathcal{R} \wedge \text{funas-gterm } u \subseteq \text{fset } \mathcal{F}\} \wedge \text{funas-gterm } t \subseteq \text{fset } \mathcal{F}\}$

definition Q-infty \mathcal{A} $\mathcal{F} = \{\{q \mid q. \text{infinite } \{t \mid t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q \in | \text{ta-der } \mathcal{A} (\text{term-of-gterm } (\text{gterm-to-None-Some } t))\}\}\}$

lemma Q-infty-fmember:
q $\in |$ Q-infty \mathcal{A} $\mathcal{F} \longleftrightarrow \text{infinite } \{t \mid t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q \in | \text{ta-der } \mathcal{A} (\text{term-of-gterm } (\text{gterm-to-None-Some } t))\}$
proof –
have $\{q \mid q. \text{infinite } \{t \mid t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q \in | \text{ta-der } \mathcal{A} (\text{term-of-gterm } (\text{gterm-to-None-Some } t))\}\} \subseteq \text{fset } (\mathcal{Q} \mathcal{A})$
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**

$q\text{-inf-dash-intro-rules } Q \ r \equiv \text{if } (r\text{-rhs } r) \mid \in \mid Q \wedge \text{fst } (r\text{-root } r) = \text{None} \text{ then } \{(r\text{-root } r) \text{ (map } CInl \text{ (} r\text{-lhs-states } r)) \rightarrow CInr \text{ (} r\text{-rhs } r)\} \text{ else } \{\mid\}$

abbreviation $\text{args} :: 'a \text{ list} \Rightarrow \text{nat} \Rightarrow ('a + 'a) \text{ list}$ **where**

$\text{args} \equiv \lambda \text{ qs } i. \text{map } CInl \text{ (take } i \text{ qs)} \ @ \ CInr \text{ (qs ! } i) \ # \ \text{map } CInl \text{ (drop (Suc } i) \text{ qs)}$

abbreviation $q\text{-inf-dash-closure-rules} :: ('q, 'f) \text{ ta-rule} \Rightarrow ('q + 'q, 'f) \text{ ta-rule list}$ **where**

$q\text{-inf-dash-closure-rules } r \equiv (\text{let } (f, \text{qs}, q) = (r\text{-root } r, r\text{-lhs-states } r, r\text{-rhs } r) \text{ in } (\text{map } (\lambda i. f \text{ (args qs } i) \rightarrow CInr \text{ } q) [0 .. < \text{length qs}]$)

definition $\text{Inf-automata} :: ('q, 'f \text{ option} \times 'f \text{ option}) \text{ ta} \Rightarrow 'q \text{ fset} \Rightarrow ('q + 'q, 'f \text{ option} \times 'f \text{ option}) \text{ ta}$ **where**

$\text{Inf-automata } \mathcal{A} \ Q = \text{TA}$
 $((\mid \cup \mid (q\text{-inf-dash-intro-rules } Q \mid \uparrow \text{ rules } \mathcal{A})) \mid \cup \mid ((\text{fset-of-list} \circ q\text{-inf-dash-closure-rules}) \mid \uparrow \text{ rules } \mathcal{A})) \mid \cup \mid$
 $\text{map-ta-rule } CInl \text{ id } \mid \uparrow \text{ rules } \mathcal{A}) \text{ (map-both } Inl \mid \uparrow \text{ eps } \mathcal{A} \mid \cup \mid \text{map-both } CInr \mid \uparrow \text{ eps } \mathcal{A})$

definition Inf-reg **where**

$\text{Inf-reg } \mathcal{A} \ Q = \text{Reg } (CInr \mid \uparrow \text{ fin } \mathcal{A}) \text{ (Inf-automata (ta } \mathcal{A}) \ Q)$

lemma Inr-Inl-rel-comp :

$\text{map-both } CInr \mid \uparrow \text{ } S \mid O \mid \text{map-both } CInl \mid \uparrow \text{ } S = \{\mid\}$ **by auto**

lemmas $\text{eps-split} = \text{ftrancl-Un2-separatorE}[OF \text{ Inr-Inl-rel-comp}]$

lemma $\text{Inf-automata-eps-simp}$ $[\text{simp}]$:

shows $(\text{map-both } Inl \mid \uparrow \text{ eps } \mathcal{A} \mid \cup \mid \text{map-both } CInr \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid =$
 $(\text{map-both } CInl \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid \mid \cup \mid (\text{map-both } CInr \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid$

proof –

{fix $x \ y \ z$ **assume** $(x, y) \mid \in \mid (\text{map-both } CInl \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid$
 $(y, z) \mid \in \mid (\text{map-both } CInr \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid$

then have False

by $(\text{metis } Inl\text{-Inr}\text{-False } \text{eps-statesI}(1, 2) \text{ eps-states-image } \text{fimageE } \text{ftranclD } \text{ftranclD2})$

then show $?thesis$ **by** $(\text{auto simp: Inf-automata-def eps-split})$

qed

lemma $\text{map-both-CInl-ftrancl-conv}$:

$(\text{map-both } CInl \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid = \text{map-both } CInl \mid \uparrow (\text{eps } \mathcal{A}) \mid + \mid$
by $(\text{intro } \text{ftrancl-map-both-fsubset}) \text{ (auto simp: finj-CInl-CInr)}$

lemma $\text{map-both-CInr-ftrancl-conv}$:

$(\text{map-both } CInr \mid \uparrow \text{ eps } \mathcal{A}) \mid + \mid = \text{map-both } CInr \mid \uparrow (\text{eps } \mathcal{A}) \mid + \mid$
by $(\text{intro } \text{ftrancl-map-both-fsubset}) \text{ (auto simp: finj-CInl-CInr)}$

lemmas $\text{map-both-ftrancl-conv} = \text{map-both-CInl-ftrancl-conv } \text{map-both-CInr-ftrancl-conv}$

lemma *Inf-automata-Inl-to-eps* [simp]:

(CInl p, CInl q) |∈| (map-both CInl |^q eps A)|⁺ | ↔ (p, q) |∈| (eps A)|⁺
(CInr p, CInr q) |∈| (map-both CInr |^q eps A)|⁺ | ↔ (p, q) |∈| (eps A)|⁺
(CInl q, CInl p) |∈| (map-both CInr |^q eps A)|⁺ | ↔ False
(CInr q, CInr p) |∈| (map-both CInl |^q eps A)|⁺ | ↔ False
by (auto simp: map-both-ftrancl-conv dest: fmap-prod-fimageI)

lemma *Inl-eps-Inr*:

(CInl q, CInl p) |∈| (eps (Inf-automata A Q))|⁺ | ↔ (CInr q, CInr p) |∈| (eps (Inf-automata A Q))|⁺
by (auto simp: Inf-automata-def)

lemma *Inr-rhs-eps-Inr-lhs*:

assumes (q, CInr p) |∈| (eps (Inf-automata 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 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 A Q))|⁺ | ↔ False
(CInr q, CInl p) |∈| (eps (Inf-automata 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 A) t |⊆| *ta-der* (Inf-automata A Q) t
proof (rule *ta-der-mono*)
show *rules* (fmap-states-ta CInl A) |⊆| *rules* (Inf-automata 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 A) |⊆| *eps* (Inf-automata A Q)
by (rule *fsubsetI*) (simp add: Inf-automata-def fmap-states-ta-def')
qed

lemma *Inl-res-A-res-Inf-automata*:

CInl |^q *ta-der* A (term-of-gterm t) |⊆| *ta-der* (Inf-automata A Q) (term-of-gterm t)
by (intro *fsubset-trans*[OF *ta-der-fmap-states-ta-mono*[of CInl A t]]) (auto simp: Inl-A-res-Inf-automata)

lemma *r-rhs-CInl-args-A-rule*:

assumes f qs → CInl q |∈| *rules* (Inf-automata A Q)
obtains qs' **where** qs = map CInl qs' f qs' → q |∈| *rules* A **using** *assms*
by (auto simp: Inf-automata-def split!: if-splits)

lemma *A-rule-to-dash-closure*:

assumes $f\ qs \rightarrow q \mid \in \mid$ *rules* \mathcal{A} **and** $i < \text{length } qs$
shows $f\ (\text{args } qs\ i) \rightarrow \text{CInr } q \mid \in \mid$ *rules* $(\text{Inf-automata } \mathcal{A}\ Q)$
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 $\text{CInl } p \mid \in \mid$ *ta-der* $(\text{Inf-automata } \mathcal{A}\ Q)$ $C\langle \text{Var } (\text{CInl } q) \rangle$
shows $\text{CInr } p \mid \in \mid$ *ta-der* $(\text{Inf-automata } \mathcal{A}\ Q)$ $C\langle \text{Var } (\text{CInr } q) \rangle$ (**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 → q' | ∈ | rules ?A length qs = Suc (length ss + length ts) and
eps: q' = CInl p ∨ (q', CInl p) | ∈ | (eps ?A)⁺ and
reach: ∀ i < Suc (length ss + length ts). qs ! i | ∈ | ta-der ?A ((ss @ C⟨Var
(CInl q)⟩ # ts) ! i)
by *auto*
from *eps* **obtain** q'' **where** [*simp*]: $q' = \text{CInl } q''$
by (*cases q'*) (*auto simp add: Inf-automata-def eps-split elim: ftranclE con-*
verse-ftranclE)
from *rule* **obtain** qs' **where** *args: qs = map CInl qs' f qs' → q'' | ∈ | rules A*
using *r-rhs-CInl-args-A-rule* **by** (*metis q' = CInl q''*)
then have $\text{CInl } (qs' ! \text{length } ss) \mid \in \mid$ *ta-der* $(\text{Inf-automata } \mathcal{A}\ Q)$ $C\langle \text{Var } (\text{CInl } q) \rangle$
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 $\text{run } \mathcal{A}\ r$ (*gterm-to-None-Some t*) **and** *ex-rule-state r | ∈ | Q*
shows $\text{CInr } (\text{ex-rule-state } r) \mid \in \mid$ *gta-der* $(\text{Inf-automata } \mathcal{A}\ Q)$ (*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)
 $\mid \in \mid$ *rules A length (gargs r) = length ts and*
reach: ∀ i < length ts. ex-comp-state (gargs r ! i) | ∈ | ta-der 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* \circ *ex-comp-state*) (*gargs* *r*)) \rightarrow *CInr* (*ex-rule-state* *r*) $|\in|$ 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 \circ ex-comp-state) (gargs r)]*)
blast
qed

lemma *Inf-automata-dash-reach-to-reach*:

assumes $p |\in|$ *ta-der* (*Inf-automata* \mathcal{A} *Q*) *t* (**is** - $|\in|$ *ta-der* ?*A* -)
shows *remove-sum* $p |\in|$ *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 \rightarrow *q'* $|\in|$ rules ?*A* *length qs = length ss* **and**
eps: q' = p \vee (*q', p*) $|\in|$ (*eps* ?*A*) $^{+}$ **and**
*reach: $\forall i < \text{length } ss. qs ! i |\in|$ ta-der ?*A* (*ss ! i*)* **by** *auto*
from rule **have** *r: f* (map (*remove-sum*) *qs*) \rightarrow (*remove-sum* *q'*) $|\in|$ 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* $p \vee$ (*remove-sum* *q'*, *remove-sum* p) $|\in|$ (*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 $\exists p q. p @ q \in$ *gposs* *u* $\wedge n < \text{length } q \wedge p \notin$ *gposs* *t*
proof -
from *poss-length-depth* **obtain** $p m$ **where** $p \in$ *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 $n < \text{length } (\text{drop } (\text{Suc } m') \ p)$ **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 A R* **and** $t \in \text{Inf-branching-terms } \mathcal{R} \ \mathcal{F}$
shows $\exists u. \text{gpair } t \ u \in \mathcal{L} \ (\text{Inf-reg } \mathcal{A} \ (\mathcal{Q}\text{-infty } (\text{ta } \mathcal{A}) \ \mathcal{F}))$ (**is** $\exists u. \text{gpair } t \ u \in \mathcal{L} \ ?B$)

proof –

let $?A = \text{Inf-automata } (\text{ta } \mathcal{A}) \ (\mathcal{Q}\text{-infty } (\text{ta } \mathcal{A}) \ \mathcal{F})$
let $?t\text{-of-}g = \lambda t. \text{term-of-gterm } t :: ('b, 'a) \text{ term}$
obtain n **where** *depth-card: depth (?t-of-g t) + fcard (Q (ta A)) < n* **by** *auto*
from *assms(1, 2)* **have** *fin: infinite {u. gpair t u ∈ L A ∧ funas-gterm u ⊆ fset F}*

by (*auto simp: RR2-spec-def Inf-branching-terms-def*)

from *infinte-no-depth-limit[of ?t-of-g ‘ {u. gpair t u ∈ L A ∧ funas-gterm u ⊆ fset F} fset F] this*

have $\forall n. \exists t \in ?t\text{-of-}g \ ‘ \{u. \text{gpair } t \ u \in \mathcal{L} \ \mathcal{A} \wedge \text{funas-gterm } u \subseteq \text{fset } \mathcal{F}\}. n < \text{depth } t$

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 ⊆ fset F* **and**

depth: Suc n < depth (?t-of-g u) **and** *lang: gpair t u ∈ L A* **by** *auto*

have $\text{Suc } (\text{depth } (\text{term-of-gterm } t) + \text{fcard } (\mathcal{Q} \ (\text{ta } \mathcal{A}))) < \text{depth } (\text{term-of-gterm } u)$

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 ∈ gposs u $p \notin \text{gposs } t$ **and** *card: fcard (Q (ta 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** $r: \text{run } (\text{ta } \mathcal{A}) \ r \ (\text{gpair } t \ u) \ \text{ex-comp-state } r \ |\in| \ \text{fin } \mathcal{A}$

unfolding *L-def gta-lang-def* **by** (*fastforce dest: gta-der-to-run*)

from *pos* **have** $p\text{-gtu}: p \in \text{gposs } (\text{gpair } t \ u)$ **and** $pu: p \in \text{gposs } u$

using *not-gposs-append* **by** *auto*

have *qinf: ex-rule-state (gsubt-at r p) |\in| Q-infty (ta A) 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-infinet-terms[of ta A gsubt-at (gpair t u) p -

λ t. t ⊆ (λ(f, n). ((None, Some f), n)) ‘ fset 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)) |\in| gta-der (Inf-automata (ta A) (Q-infty (ta A) F)) (gsubt-at (gpair t u) p)*

unfolding $gp[symmetric]$.
have $CInl (ex-comp-state\ r) \mid \in \mid ta-der\ ?A (ctxt-of-pos-term\ p (term-of-gterm (gpair\ t\ u))) \langle Var\ (CInl\ (ex-rule-state\ (gsubt-at\ r\ p))) \rangle$
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 $CInr (ex-comp-state\ r) \mid \in \mid gta-der\ (Inf-automata\ (ta\ \mathcal{A})\ (Q-infty\ (ta\ \mathcal{A})\ \mathcal{F})) (gpair\ t\ u)$
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** $CInr (ex-comp-state\ r) \mid \in \mid fin\ (Inf-reg\ \mathcal{A}\ (Q-infty\ (ta\ \mathcal{A})\ \mathcal{F}))$
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 $CInr\ p \mid \in \mid ta-der\ (Inf-automata\ \mathcal{A}\ Q)\ t$ **and** $ground\ t$
shows $\exists\ C\ s\ q. C \langle s \rangle = t \wedge CInr\ q \mid \in \mid ta-der\ (Inf-automata\ \mathcal{A}\ Q)\ s \wedge q \mid \in \mid Q \wedge$
 $CInr\ p \mid \in \mid ta-der\ (Inf-automata\ \mathcal{A}\ Q)\ C \langle Var\ (CInr\ q) \rangle \wedge$
 $(fst \circ fst \circ the \circ root)\ s = None$ **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**
 $rule: f\ qs \rightarrow CInr\ q' \mid \in \mid rules\ ?A\ length\ qs = length\ ts$ **and**
 $eps: q' = p \vee (CInr\ q', CInr\ p) \mid \in \mid (eps\ ?A)^+ \mid$ **and**
 $reach: \forall\ i < length\ ts. qs\ !\ i \mid \in \mid ta-der\ ?A\ (ts\ !\ i)$
by $auto\ (metis\ Inr-rhs-eps-Inr-lhs)$
consider $(a) \wedge i. i < length\ qs \implies \exists\ q''. qs\ !\ i = CInl\ q'' \mid (b) \exists\ i < length\ qs. \exists\ q''. qs\ !\ i = CInr\ q''$
by $(meson\ remove-sum.cases)$
then show $?case$
proof $cases$
case a
then have $f\ qs \rightarrow CInr\ q' \mid \in \mid \bigcup \mid (q-inf-dash-intro-rules\ Q\ \mid^i\ rules\ \mathcal{A})$ **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'' |∈| ta-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⟨s⟩ = ts ! i and
 qinf: CInr q''' |∈| ta-der (Inf-automata A Q) s ∧ q''' |∈| Q and
 reach2: CInr q'' |∈| ta-der (Inf-automata A Q) C⟨Var (CInr q''')⟩ and
 (fst ∘ fst ∘ the ∘ 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 A R and $\mathcal{R} \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \times \mathcal{T}_G (\text{fset } \mathcal{F})$
 and infinite {u | u. gpair t u ∈ L A}
 shows t ∈ Inf-branching-terms R F
 proof -
 from assms have [simp]: gpair t u ∈ L A \longleftrightarrow (t, u) ∈ R ∧ u ∈ $\mathcal{T}_G (\text{fset } \mathcal{F})$
 for u by (auto simp: RR2-spec-def)
 from assms have t ∈ $\mathcal{T}_G (\text{fset } \mathcal{F})$ 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 A R and $\mathcal{R} \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \times \mathcal{T}_G (\text{fset } \mathcal{F})$
 and gpair t u ∈ L (Inf-reg A (Q-infty (ta A) F))
 shows t ∈ Inf-branching-terms R F
 proof -
 let ?con = λ t. term-of-gterm (gterm-to-None-Some t)
 let ?A = Inf-automata (ta A) (Q-infty (ta A) F)
 from assms(3) obtain q where fin: q |∈| fin A and
 reach-fin: CInr q |∈| ta-der ?A (term-of-gterm (gpair t u))
 by (auto simp: Inf-reg-def 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⟨s⟩ = term-of-gterm (gpair t u) and
 q-inf-st: CInr p |∈| ta-der ?A s p |∈| Q-infty (ta A) F and
 reach: CInr q |∈| ta-der ?A C⟨Var (CInr p)⟩ and

```

  none: (fst ◦ fst ◦ the ◦ root) s = None by auto
have gr: ground s ground-ctxt C using arg-cong[OF ctxt, of ground]
  by auto
have reach: q |∈| ta-der (ta A) (adapt-vars-ctxt C)⟨Var p⟩
  using gr Inf-automata-dash-reach-to-reach[OF reach]
  by (auto simp: map-vars-term-ctxt-commute)
from q-inf-st(2) have inf: infinite {v. funas-gterm v ⊆ fset F ∧ p |∈| ta-der (ta
A) (?con v)}
  by (simp add: Q-infty-fmember)
have inf: infinite {v. funas-gterm v ⊆ fset F ∧ q |∈| gta-der (ta A) (gctxt-of-ctxt
C)⟨gterm-to-None-Some v⟩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⟨s⟩) (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) ∉ 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))⟨v⟩G = (gctxt-of-ctxt C)⟨gterm-to-None-Some
v⟩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 ⟨ghole-pos (gctxt-of-ctxt C) = hole-pos C⟩ gfun-at-None-ngposs-iff
gfun-at-gpair gsubt-at-gctxt-apply-ghole hole-pos-poss hp-nt poss-gposs-conv
term-of-gterm-ctxt-apply)
have inf: infinite {v. gpair t ((gctxt-at-pos u (hole-pos C))⟨v⟩G) ∈ L A} using
fin
  by (intro infinite-super[OF - inf]) (auto simp: L-def gta-der-def simp flip:
to-gpair)
have infinite {u | u. gpair t u ∈ L 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 A]) simp
qed

```

lemma Inf-automata-subseteq:

$\mathcal{L} (\text{Inf-reg } \mathcal{A} (Q\text{-infty } (ta \mathcal{A}) \mathcal{F})) \subseteq \mathcal{L} \mathcal{A}$ (is $\mathcal{L} ?IA \subseteq -$)

proof

fix s **assume** l: s ∈ L ?IA

then obtain q **where** w: q |∈| fin ?IA q |∈| ta-der (ta ?IA) (term-of-gterm s)

by (auto simp: L-def elim!: gta-langE)

```

from  $w(1)$  have remove-sum  $q \in | \text{fin } \mathcal{A}$ 
  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  $q \text{ ta } \mathcal{A}$ ]  $w(2)$ 
  by (auto simp: gterm.map-ident L-def Inf-reg-def)
    (metis gta-langI map-vars-term-term-of-gterm)
qed

```

lemma *L-Inf-reg*:

```

assumes RR2-spec  $\mathcal{A} \mathcal{R}$  and  $\mathcal{R} \subseteq \mathcal{T}_G (\text{fset } \mathcal{F}) \times \mathcal{T}_G (\text{fset } \mathcal{F})$ 
shows gfst ‘  $\mathcal{L} (\text{Inf-reg } \mathcal{A} (Q\text{-infty } (\text{ta } \mathcal{A}) \mathcal{F})) = \text{Inf-branching-terms } \mathcal{R} \mathcal{F}$ 

```

proof –

```

{fix  $s$  assume ass:  $s \in \mathcal{L} (\text{Inf-reg } \mathcal{A} (Q\text{-infty } (\text{ta } \mathcal{A}) \mathcal{F}))$ 
  then have  $\exists t u. s = \text{gpair } t u$  using Inf-automata-subseteq[of  $\mathcal{A} \mathcal{F}$ ] assms(1)
    by (auto simp: RR2-spec-def)
  then have  $\text{gfst } s \in \text{Inf-branching-terms } \mathcal{R} \mathcal{F}$ 
    using ass Inf-automata-to-Inf[OF assms]
    by (force simp: gfst-gpair)}
then show ?thesis using Inf-to-automata[OF assms(1), of - 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*]:

```

 $\text{ta-der } \mathcal{A} (\text{Var } q) = \text{finsert } q ((\text{eps } \mathcal{A})^+ | | \text{' } \{q\})$ 
by (auto)

```

lemma *ta-der-Fun-code* [*code*]:

```

 $\text{ta-der } \mathcal{A} (\text{Fun } f \text{ ts}) =$ 
  (let args = map (ta-der  $\mathcal{A}$ ) ts in
    let  $P = (\lambda r. \text{case } r \text{ of } \text{TA-rule } g \text{ ps } p \Rightarrow f = g \wedge \text{list-all2 } \text{fmember } \text{ps } \text{args})$  in
      let  $S = \text{r-rhs } | \text{' } \text{ffilter } P (\text{rules } \mathcal{A})$  in
         $S \cup (\text{eps } \mathcal{A})^+ | | \text{' } S$  (is  $?Ls = ?Rs$ )

```

proof

```

{fix  $q$  assume  $q \in | ?Ls$  then have  $q \in | ?Rs$ 
  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  $?Ls \subseteq | ?Rs$  by blast
next
  {fix  $q$  assume  $q \in | ?Rs$  then have  $q \in | ?Ls$ 
    apply (auto simp: Let-def fBex-def list-all2-conv-all-nth
      fImage-iff)
  }

```

```

      split!: ta-rule.splits)
    apply (metis ta-rule.collapse)
    apply blast
  done}
  then show ?Rs  $\subseteq$  ?Ls by blast
qed

```

definition *eps-free-automata* **where**

```

  eps-free-automata epscl  $\mathcal{A}$  =
    (let ruleps = ( $\lambda$  r. finsert (r-rhs r) (epscl  $\{|r-rhs\ r|\}$ )) in
     let rules = ( $\lambda$  r. ( $\lambda$  q. TA-rule (r-root r) (r-lhs-states r) q)  $\{|ruleps\ r|\}$ 
      (rules  $\mathcal{A}$ ) in
      TA ( $\bigcup$  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  $\mathcal{A}$  =  $\{\|\}$   $\implies$  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 ( $\lambda$  r. case r of TA-rule f ps p  $\implies$  fset-of-list ps  $\subseteq$   $Q$   $\wedge$  p
   $\in$   $Q$ ) (rules  $\mathcal{A}$ ) in
     let eps = ffilter ( $\lambda$  r. case r of (p, q)  $\implies$  p  $\in$   $Q$   $\wedge$  q  $\in$   $Q$ ) (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]:

$fCollect (prod-epsLp \mathcal{A} \mathcal{B}) = (\lambda ((p, q), r). ((p, r), (q, r))) \upharpoonright^q (eps \mathcal{A} \times \mathcal{Q} \mathcal{B})$
 $fCollect (prod-epsRp \mathcal{A} \mathcal{B}) = (\lambda ((p, q), r). ((r, p), (r, q))) \upharpoonright^q (eps \mathcal{B} \times \mathcal{Q} \mathcal{A})$
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*]: $f qs \rightarrow q \mid \in \mid rules \mathcal{A} \implies \forall i < length\ qs. qs ! i \in ta-reach \mathcal{A} \implies q \in ta-reach \mathcal{A}$
 $\mid eps$ [*intro*]: $q \in ta-reach \mathcal{A} \implies (q, r) \mid \in \mid eps \mathcal{A} \implies r \in ta-reach \mathcal{A}$

lemma *ta-reach-eps-transI*:

assumes $(p, q) \mid \in \mid (eps \mathcal{A}) \upharpoonright^+ \mid p \in ta-reach \mathcal{A}$
shows $q \in ta-reach \mathcal{A}$ **using** *assms*
by (*induct rule: ftrancl-induct*) *auto*

lemma *ta-reach-ground-term-der*:

assumes $q \in ta-reach \mathcal{A}$
shows $\exists t. ground\ t \wedge q \mid \in \mid ta-der \mathcal{A} t$ **using** *assms*

proof (*induct*)

case (*rule* $f qs q$)

then obtain *ts* **where** $length\ ts = length\ qs$

$\forall i < length\ qs. ground\ (ts ! i)$

$\forall i < length\ qs. qs ! i \mid \in \mid ta-der \mathcal{A} (ts ! i)$

using *Ex-list-of-length-P*[*of length qs λ t i. ground t ∧ qs ! i ∣ ∈ ∣ ta-der 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 $ground\ t\ q \mid \in \mid ta-der \mathcal{A} t$

shows $q \in ta-reach \mathcal{A}$ **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 - A*]

using *ground-term-der-ta-reach*[*of - - A*]

unfolding *ta-reachable-def*

by *auto*

9.1 Horn setup for reachable states

definition *reach-rules* $\mathcal{A} =$

$\{qs \rightarrow_h q \mid f qs q. TA-rule\ f\ qs\ q \mid \in \mid rules \mathcal{A}\} \cup$

```

    {[q] →h r | q r. (q, r) |∈| eps  $\mathcal{A}$ }

locale reach-horn =
  fixes  $\mathcal{A} :: ('q, 'f)$  ta
begin

sublocale horn reach-rules  $\mathcal{A}$  .

lemma reach-infer0: infer0 = {q | f q. TA-rule f [] q |∈| rules  $\mathcal{A}$ }
  by (auto simp: horn.infer0-def reach-rules-def)

lemma reach-infer1:
  infer1 p X = {r | f qs r. TA-rule f qs r |∈| rules  $\mathcal{A} \wedge p \in \text{set } qs \wedge \text{set } qs \subseteq \text{insert } p X\} \cup
  \{r | r. (p, r) |∈| 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 ⇒ ('q,'f) ta ⇒ 'q set for P and  $\mathcal{A} ::$ 
('q,'f) ta where
  basic [intro]: q |∈| P ⇒ q ∈ ta-productive-ind P  $\mathcal{A}$ 
  | eps [intro]: (p, q) |∈| (eps  $\mathcal{A}$ )+ ⇒ q ∈ ta-productive-ind P  $\mathcal{A}$  ⇒ p ∈ ta-productive-ind
P  $\mathcal{A}$ 
  | rule: TA-rule f qs q |∈| rules  $\mathcal{A}$  ⇒ q ∈ ta-productive-ind P  $\mathcal{A}$  ⇒ q' ∈ set qs
⇒ q' ∈ ta-productive-ind P  $\mathcal{A}$ 

```

```

lemma ta-productive-ind:
  ta-productive-ind  $P \mathcal{A} = \text{fset } (ta-productive \ P \ \mathcal{A})$  (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 :  $r \in | \in | \ ta\text{-der } \mathcal{A} \ C \langle \text{Var } q \rangle$  and  $f$  :  $r \in | \in | \ P$  by auto
    from  $f$  have  $r \in ta-productive-ind \ P \ \mathcal{A} \ r \in | \in | \ ta-productive \ P \ \mathcal{A}$ 
      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 :  $f \ ps \rightarrow p \in | \in | \ rules \ \mathcal{A} \ p = r \vee (p, r) \in | \in | \ (eps \ \mathcal{A})^+ \ | \ length \ ps = length$ 
      (ss @ C <Var q> # ts)
       $ps \ ! \ length \ ss \in | \in | \ ta\text{-der } \mathcal{A} \ C \langle \text{Var } q \rangle$ 
      by (auto simp: nth-append-Cons split: if-splits)
    then have  $p \in ta-productive-ind \ P \ \mathcal{A} \implies p \in | \in | \ ta\text{-der } \mathcal{A} \ C \langle \text{Var } q \rangle \implies q \in$ 
    ta-productive-ind P 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  $P \mathcal{A} = \{[] \rightarrow_h q \mid q. q \in | \in | \ P\} \cup$ 
   $\{[r] \rightarrow_h q \mid q \ r. (q, r) \in | \in | \ eps \ \mathcal{A}\} \cup$ 
   $\{[q] \rightarrow_h r \mid f \ qs \ q \ r. TA\text{-rule } f \ qs \ q \in | \in | \ rules \ \mathcal{A} \wedge r \in set \ qs\}$ 

```

```

locale productive-horn =
  fixes  $\mathcal{A} :: ('q, 'f) \ ta$  and  $P :: 'q \ fset$ 
begin

```

```

sublocale horn productive-rules P 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 A} ∪
    {r | f qs r. TA-rule f qs p |∈| rules 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 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  $p: ts[i := p] \in lists (insert\ p\ X)$ 
  using set-update-subset-insert by fastforce
then have  $p \in set (ts[i := p]) \map fst (ts[i := p]) = qs \ set (ts[i := p]) \subseteq insert\ p\ X$ 
  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 \rightarrow_h (Wrapp\ q) \mid rs\ f\ q.\$ 
   $q = ps\text{-reachable-states}\ \mathcal{A}\ f\ (\map\ ex\ rs) \wedge q \neq \{\|\}\}$ 

```

```

locale ps-states-horn =
  fixes  $\mathcal{A} :: ('q, 'f)\ ta$ 
begin

```

```

sublocale horn ps-states-rules  $\mathcal{A}$  .

```

```

lemma ps-construction-infer0: infer0 =
   $\{Wrapp\ q \mid f\ q.\ q = ps\text{-reachable-states}\ \mathcal{A}\ f\ \|\ \wedge q \neq \{\|\}\}$ 
  by (auto simp: ps-states-rules-def horn.infer0-def)

```

```

lemma ps-construction-infer1:
  infer1  $p\ X = \{Wrapp\ q \mid f\ qs\ q.\ q = ps\text{-reachable-states}\ \mathcal{A}\ f\ (\map\ ex\ qs) \wedge q \neq$ 
 $\{\|\} \wedge$ 
   $p \in set\ qs \wedge set\ qs \subseteq insert\ p\ X\}$ 
  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  $ps \rightarrow_h (Wrapp (ps\text{-reachable-states}\ \mathcal{A}\ f\ (\map\ ex\ ps))) \in ps\text{-states-rules}\ \mathcal{A}$ 

```

```

  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  $q \in saturate\ q = p$  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  $\Delta \Delta_\varepsilon f ps =$   
  (let R = ffilter ( $\lambda r$ . case r of TA-rule g qs q  $\Rightarrow f = g \wedge list-all2 (|\in|) qs ps$ )  $\Delta$   
  in  
    let S = r-rhs | $\uparrow$ | R in  
    S | $\cup$ |  $\Delta_\varepsilon$ | $^+$ | | $\uparrow$ | S)
```

lemma *ps-reachable-states* [*code*]:

```
ps-reachable-states (TA  $\Delta \Delta_\varepsilon$ ) f ps = ps-reachable-states-cont  $\Delta \Delta_\varepsilon f ps$   
by (auto simp: ps-reachable-states-fmember ps-reachable-states-cont-def Let-def  
image-iff fBex-def  
  split!: ta-rule.splits) force+
```

definition *ps-rules-cont* **where**

```
ps-rules-cont  $\mathcal{A} Q =$   
  (let sig = ta-sig  $\mathcal{A}$  in  
    let qss = ( $\lambda (f, n)$ . (f, n, fset-of-list (List.n-lists n (sorted-list-of-fset Q)))) | $\uparrow$ |  
sig in  
    let res = ( $\lambda (f, n, Qs)$ . ( $\lambda qs$ . TA-rule f qs (Wrapp (ps-reachable-states  $\mathcal{A} f$  (map  
ex qs)))) | $\uparrow$ | Qs) | $\uparrow$ | qss in  
    ffilter ( $\lambda r$ . ex (r-rhs r)  $\neq \{\{\}\}$ ) (| $\cup$ | res))
```

lemma *ps-rules* [*code*]:

```
ps-rules  $\mathcal{A} Q = ps-rules-cont \mathcal{A} Q$   
using ps-reachable-states-sig finite-ps-rulesp-unfolded[of  $Q \mathcal{A}$ ]  
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  
  First-Order-Terms.Term-Impl
```

begin

```
derive linorder ta-rule  
derive (compare) compare 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 comparator-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  $\Rightarrow$  'a FSet-Lex-Wrapper
 $\Rightarrow$  order) option
where ccompare-FSet-Lex-Wrapper = Some ( $\lambda$  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 comparator-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 =  $\lambda$  n. (f (replicate n q)  $\rightarrow$  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 = ( $\lambda$  ( - :: ('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 ⊆ insert x F ∧ finite X
      then have X - {x} ⊆ F using ass by auto
      then have fpow : X - {x} ∈ Fpow F using conjunct2[OF ass]
        by (auto simp: Fpow-def)
      have X ∈ Fpow F ∪ insert x ‘ Fpow F
      proof (cases x ∈ X)
        case True
          then have X ∈ 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 ∪ 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 ⊆ 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:
  (∧ X. X ∈ A ⇒ finite X) ⇒ 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 ∘ 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* \circ *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* \circ *Abs-fset*) ' *?FP*) \implies *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* \Rightarrow 'a *FSet-Lex-Wrapper*
option \Rightarrow bool **where**

cproper-interval-FSet-Lex-Wrapper *None* *None* \longleftrightarrow *True*

| *cproper-interval-FSet-Lex-Wrapper* *None* (*Some* *B*) \longleftrightarrow (\exists *Z*. *sorted-list-of-fset*
(*ex* *Z*) < *sorted-list-of-fset* (*ex* *B*))

| *cproper-interval-FSet-Lex-Wrapper* (*Some* *A*) *None* \longleftrightarrow (\exists *Z*. *sorted-list-of-fset*
(*ex* *A*) < *sorted-list-of-fset* (*ex* *Z*))

| *cproper-interval-FSet-Lex-Wrapper* (*Some* *A*) (*Some* *B*) \longleftrightarrow (\exists *Z*. *sorted-list-of-fset*
(*ex* *A*) < *sorted-list-of-fset* (*ex* *Z*) \wedge

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  $\implies$  lt-of-comp f X Z  $\longleftrightarrow$  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 =  $\lambda n. Fun f (replicate n (Var v))$ 
  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 cpproper-interval = ( $\lambda ( - :: ('a,'b)term option) - . False$ )
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  $\Rightarrow$  'a)  $\Rightarrow$  ('b  $\Rightarrow$  'c list)  $\Rightarrow$  'b list  $\Rightarrow$  ('a, 'c list)
mapping where

```

$map\text{-}val\text{-}of\text{-}list\ ek\ ev\ xs = foldr\ (\lambda x\ m.\ Mapping.update\ (ek\ x)\ (ev\ x\ @\ case\ option\ Nil\ id\ (Mapping.lookup\ m\ (ek\ x))))\ m\ xs\ Mapping.empty$

abbreviation $map\text{-}of\text{-}list\ ek\ ev\ xs \equiv map\text{-}val\text{-}of\text{-}list\ ek\ (\lambda x.\ [ev\ x])\ xs$

lemma $map\text{-}val\text{-}of\text{-}list\text{-}tabulate\text{-}conv$:

$map\text{-}val\text{-}of\text{-}list\ ek\ ev\ xs = Mapping.tabulate\ (sort\ (remdups\ (map\ ek\ xs)))\ (\lambda k.\ concat\ (map\ ev\ (filter\ (\lambda x.\ k = ek\ x)\ xs)))$

unfolding $map\text{-}val\text{-}of\text{-}list\text{-}def$

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 $map\text{-}val\text{-}of\text{-}list\text{-}simp = map\text{-}val\text{-}of\text{-}list\text{-}tabulate\text{-}conv\ lookup\ tabulate$

9.4 Setup for the list implementation of reachable states

definition $reach\text{-}infer0\text{-}cont$ **where**

$reach\text{-}infer0\text{-}cont\ \Delta =$
 $map\ r\text{-}rhs\ (filter\ (\lambda r.\ case\ r\ of\ TA\text{-}rule\ f\ ps\ p \Rightarrow ps = [])\ (sorted\text{-}list\text{-}of\text{-}fset\ \Delta))$

definition $reach\text{-}infer1\text{-}cont :: ('q :: linorder, 'f :: linorder)\ ta\text{-}rule\ fset \Rightarrow ('q \times 'q)\ fset \Rightarrow 'q \Rightarrow 'q\ fset \Rightarrow 'q\ list$ **where**

$reach\text{-}infer1\text{-}cont\ \Delta\ \Delta_\epsilon =$
 $(let\ rules = sorted\text{-}list\text{-}of\text{-}fset\ \Delta\ in$
 $let\ eps = sorted\text{-}list\text{-}of\text{-}fset\ \Delta_\epsilon\ in$
 $let\ mapp\text{-}r = map\text{-}val\text{-}of\text{-}list\ fst\ snd\ (concat\ (map\ (\lambda r.\ map\ (\lambda q.\ (q,\ [r])))\ (r\text{-}lhs\text{-}states\ r))\ rules)\ in$
 $let\ mapp\text{-}e = map\text{-}of\text{-}list\ fst\ snd\ eps\ in$
 $(\lambda p\ bs.$
 $(map\ r\text{-}rhs\ (filter\ (\lambda r.\ case\ r\ of\ TA\text{-}rule\ f\ qs\ q \Rightarrow$
 $fset\text{-}of\text{-}list\ qs\ \subseteq\ finsert\ p\ bs)\ (case\ option\ Nil\ id\ (Mapping.lookup\ mapp\text{-}r$
 $p))))\ @$
 $case\ option\ Nil\ id\ (Mapping.lookup\ mapp\text{-}e\ p)))$

locale $reach\text{-}rules\text{-}fset =$

fixes $\Delta :: ('q :: linorder, 'f :: linorder)\ ta\text{-}rule\ fset$ **and** $\Delta_\epsilon :: ('q \times 'q)\ fset$
begin

sublocale $reach\text{-}horn\ TA\ \Delta\ \Delta_\epsilon .$

lemma $infer1$:

$infer1\ p\ (fset\ bs) = set\ (reach\text{-}infer1\text{-}cont\ \Delta\ \Delta_\epsilon\ p\ bs)$

unfolding $reach\text{-}infer1\ reach\text{-}infer1\text{-}cont\text{-}def\ set\ append\ Un\text{-}assoc[symmetric]\ Let\text{-}def$

unfolding $sorted\text{-}list\text{-}of\text{-}fset\text{-}simps\ union\text{-}fset$

apply ($intro\ arg\ cong2[of\ -\ -\ -\ (\cup)]$)

```

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 Δε) reach-infer0-cont Δ reach-infer1-cont
  Δ Δε
  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 Δε =
  horn-fset-impl.saturate-impl (reach-infer0-cont Δ) (reach-infer1-cont Δ Δε)

lemma reach-fset-impl-sound:
  reach-cont-impl Δ Δε = Some xs ⇒ fset xs = ta-reach (TA Δ Δε)
  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 Δε ≠ None
proof –
  have finite (ta-reach (TA Δ Δε))
  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* \Rightarrow ('q \times 'q) *fset* \Rightarrow 'q \Rightarrow 'q *fset* \Rightarrow '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 \times '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 - - - (\cup)])
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* $P \Delta \Delta_\varepsilon =$
horn-fset-impl.saturate-impl (*sorted-list-of-fset* P) (*productive-infer1-cont* $\Delta \Delta_\varepsilon$)

lemma *productive-cont-impl-sound*:

productive-cont-impl $P \Delta \Delta_\varepsilon = \text{Some } xs \implies \text{fset } xs = \text{ta-productive-ind } P (TA \Delta \Delta_\varepsilon)$

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 $P \Delta \Delta_\varepsilon \neq \text{None}$

proof –

have *finite* (*ta-productive-ind* $P (TA \Delta \Delta_\varepsilon)$)

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 $P (TA \Delta \Delta_\varepsilon) = \text{the } (\text{productive-cont-impl } P \Delta \Delta_\varepsilon)$

using *productive-cont-impl-complete*[*of* $P \Delta$] *productive-cont-impl-sound*[*of* $P \Delta$]

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* $r \equiv \text{length } (r\text{-lhs-states } r)$

definition *ps-reachable-states-list* **where**

ps-reachable-states-list *mapp-r* *mapp-e* *f* *ps* =

(*let* $R = \text{filter } (\lambda r. \text{list-all2 } (|\in|) (r\text{-lhs-states } r) \text{ps})$

(*case-option* *Nil id* (*Mapping.lookup* *mapp-r* (*f*, *length* *ps*))) *in*

let $S = \text{map } r\text{-rhs } R$ *in*

$S @ \text{concat } (\text{map } (\text{case-option } \text{Nil id} \circ \text{Mapping.lookup } \text{mapp-e}) S)$)

lemma *ps-reachable-states-list-sound*:

assumes *length* *ps* = n

and *mapp-r*: *case-option* *Nil id* (*Mapping.lookup* *mapp-r* (*f*, n)) =

filter ($\lambda r. r\text{-root } r = f \wedge r\text{-statesl } r = n$) (*sorted-list-of-fset* Δ)

and *mapp-e*: $\bigwedge p. \text{case-option } \text{Nil id } (\text{Mapping.lookup } \text{mapp-e } p) =$

map *snd* (*filter* ($\lambda q. \text{fst } q = p$) (*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 ($TA \Delta \Delta_\varepsilon$) *f* (*map* *ex* *ps*) (**is** $?Ls = ?Rs$)

proof –

have $*$: *length* *ps* = n *length* (*map* *ex* *ps*) = n **using** *assms* **by** *auto*

{**fix** q **assume** $q \in |$ $?Ls$ }

then obtain $qs\ p$ **where** $TA\text{-rule}\ f\ qs\ p\ |\in|\ \Delta\ length\ ps = length\ qs$
 $list\text{-all2}\ (|\in|)\ qs\ (map\ ex\ ps)\ p = q \vee (p, q)\ |\in|\ \Delta_\varepsilon|^+$
unfolding $ps\text{-reachable-states-list-def}\ Let\text{-def}\ comp\text{-def}\ assms(1, 2, 3) *$
by $(force\ simp\ add:\ fset\text{-of-list-elem}\ image\text{-iff}\ fBex\text{-def})$
then have $q\ |\in|\ ?Rs$
by $(force\ simp\ add:\ ps\text{-reachable-states-fmember}\ image\text{-iff})$
moreover
{fix q **assume** $q\ |\in|\ ?Rs$
then obtain $qs\ p$ **where** $TA\text{-rule}\ f\ qs\ p\ |\in|\ \Delta\ length\ ps = length\ qs$
 $list\text{-all2}\ (|\in|)\ qs\ (map\ ex\ ps)\ p = q \vee (p, q)\ |\in|\ \Delta_\varepsilon|^+$
by $(auto\ simp\ add:\ ps\text{-reachable-states-fmember}\ list\text{-all2-iff})$
then have $q\ |\in|\ ?Ls$
unfolding $ps\text{-reachable-states-list-def}\ Let\text{-def} * comp\text{-def}\ assms(2, 3)$
by $(force\ simp\ add:\ fset\text{-of-list-elem}\ image\text{-iff})$
ultimately show $?thesis$ **by** $blast$
qed

lemma $rule\text{-target-statesI}$:

$\exists r\ |\in|\ \Delta.\ r\text{-rhs}\ r = q \implies q\ |\in|\ rule\text{-target-states}\ \Delta$
by $auto$

definition $ps\text{-states-infer0-cont} :: ('q :: linorder, 'f :: linorder)\ ta\text{-rule}\ fset \Rightarrow$

$('q \times 'q)\ fset \Rightarrow 'q\ FSet\text{-Lex-Wrapper}\ list$ **where**

$ps\text{-states-infer0-cont}\ \Delta\ \Delta_\varepsilon =$

$(let\ sig = filter\ (\lambda\ r.\ r\text{-lhs-states}\ r = [])\ (sorted\text{-list-of-fset}\ \Delta)\ in$

$filter\ (\lambda\ p.\ ex\ p \neq \{\})\ (map\ (\lambda\ r.\ Wrapp\ (ps\text{-reachable-states}\ (TA\ \Delta\ \Delta_\varepsilon)$

$(r\text{-root}\ r)\ []))\ sig))$

definition $ps\text{-states-infer1-cont} :: ('q :: linorder, 'f :: linorder)\ ta\text{-rule}\ fset \Rightarrow ('q$
 $\times 'q)\ fset \Rightarrow$

$'q\ FSet\text{-Lex-Wrapper} \Rightarrow 'q\ FSet\text{-Lex-Wrapper}\ fset \Rightarrow 'q\ FSet\text{-Lex-Wrapper}\ list$

where

$ps\text{-states-infer1-cont}\ \Delta\ \Delta_\varepsilon =$

$(let\ sig = remdups\ (map\ (\lambda\ r.\ (r\text{-root}\ r,\ r\text{-statesl}\ r))\ (filter\ (\lambda\ r.\ r\text{-lhs-states}\ r$

$\neq [])\ (sorted\text{-list-of-fset}\ \Delta)))\ in$

$let\ arities = remdups\ (map\ snd\ sig)\ in$

$let\ etr = sorted\text{-list-of-fset}\ (\Delta_\varepsilon|^+)\ in$

$let\ mapp\text{-}r = map\text{-of-list}\ (\lambda\ r.\ (r\text{-root}\ r,\ r\text{-statesl}\ r))\ id\ (sorted\text{-list-of-fset}\ \Delta)$

in

$let\ mapp\text{-}e = map\text{-of-list}\ fst\ snd\ etr\ in$

$(\lambda\ p\ bs.$

$(let\ states = sorted\text{-list-of-fset}\ (finsert\ p\ bs)\ in$

$let\ arity\text{-to-states-map} = Mapping.\ tabulate\ arities\ (\lambda\ n.\ list\text{-of-permutation-element-}n$

$p\ n\ states)\ in$

$let\ res = map\ (\lambda\ (f,\ n).$

$map\ (\lambda\ s.\ let\ rules = the\ (Mapping.\ lookup\ mapp\text{-}r\ (f,\ n))\ in$

$Wrapp\ (fset\text{-of-list}\ (ps\text{-reachable-states-list}\ mapp\text{-}r\ mapp\text{-}e\ f\ (map\ ex\ s))))$

$(the\ (Mapping.\ lookup\ arity\text{-to-states-map}\ n)))$

```

      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 (λ r. (r-root r, r-statesl r)) (λ x. 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 (λ r. 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 Δε) (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 Δε) ps-states-infer0-cont Δ Δε ps-states-infer1-cont
Δ Δε
  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 Δε =
  horn-fset-impl.saturate-impl (ps-states-infer0-cont Δ Δε) (ps-states-infer1-cont
Δ Δε)

```

```

lemma ps-states-fset-impl-sound:
  assumes ps-states-fset-impl Δ Δε = Some xs
  shows xs = ps-states (TA Δ Δε)

```

using *ps-states-fset.saturate-impl-sound*[*OF* *assms*[*unfolded ps-states-fset-impl-def*]]
using *ps-states-horn.ps-states-sound*[*of* *TA* Δ Δ_ε]
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 Δ $\Delta_\varepsilon \neq \text{None}$

proof –

let $?R = \text{ps-states } (TA \ \Delta \ \Delta_\varepsilon)$

let $?S = \text{horn.saturate } (\text{ps-states-rules } (TA \ \Delta \ \Delta_\varepsilon))$

have $?S \subseteq \text{fset } ?R$

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* Δ Δ_ε) =

(*let* *xs* = *the* (*ps-states-fset-impl* Δ Δ_ε) *in*

TA (*ps-rules* (*TA* Δ Δ_ε) *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* Δ Δ_ε)) =

(*let* *xs* = *the* (*ps-states-fset-impl* Δ Δ_ε) *in*

Reg (*ffilter* ($\lambda S. Q \mid\cap\mid \text{ex } S \neq \{\{\}\}$) *xs*)

(*TA* (*ps-rules* (*TA* Δ Δ_ε) *xs*) $\{\{\}\}$))

using *ps-states-fset-impl-complete*[*of* Δ Δ_ε]

using *ps-states-fset-impl-sound*[*of* Δ Δ_ε]

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 ($\mathcal{A} :: ('q1 :: \text{linorder}, 'f :: \text{linorder}) \text{ta}$) ($\mathcal{B} :: ('q2 :: \text{linorder}, 'f :: \text{linorder}) \text{ta}$) =

(*let* *sig* = *sorted-list-of-fset* (*ta-sig* $\mathcal{A} \mid\cap\mid \text{ta-sig } \mathcal{B}$) *in*

let *mapA* = *map-of-list* ($\lambda r. (\text{r-root } r, \text{r-statesl } r)$) *id* (*sorted-list-of-fset* (*rules* \mathcal{A})) *in*

let *mapB* = *map-of-list* ($\lambda r. (\text{r-root } r, \text{r-statesl } r)$) *id* (*sorted-list-of-fset* (*rules* \mathcal{B})) *in*

let *merge* = ($\lambda (ra, rb). \text{TA-rule } (\text{r-root } ra) (\text{zip } (\text{r-lhs-states } ra) (\text{r-lhs-states } rb)) (\text{r-rhs } ra, \text{r-rhs } rb)$) *in*

fset-of-list (

concat (*map* ($\lambda (f, n). \text{map } \text{merge}$))

```

      (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'
∨ (q, q') |∈| (eps A)+) ∧
  length qs = length ts ∧ (∀ i < length ts. qs ! i |∈| ta-der-strict A (ts ! i))}

```

```

lemma ta-der-strict-Var:
  q |∈| ta-der-strict A (Var x) ⟷ x = q

```

unfolding *ta-der.simps* **by** *auto*

lemma *ta-der-strict-Fun*:

$q \in | \text{ta-der-strict } \mathcal{A} \text{ (Fun } f \text{ ts)} \longleftrightarrow (\exists ps \ p. \text{TA-rule } f \text{ ps } p \in | \text{(rules } \mathcal{A}) \wedge$
 $(p = q \vee (p, q) \in | \text{(eps } \mathcal{A})|^{+}) \wedge \text{length } ps = \text{length } ts \wedge$
 $(\forall i < \text{length } ts. ps \ ! \ i \in | \text{ta-der-strict } \mathcal{A} \text{ (ts } \ ! \ i))) \text{ (is } ?Ls \longleftrightarrow ?Rs)$

unfolding *ta-der-strict.simps*

by (*intro iffI fCollect-memberI finite-Collect-less-eq[OF - finite-eps[of]]*) *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-der-strict } \mathcal{A} \ t \subseteq | \text{ta-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*:

assumesground t

shows ta-der } \mathcal{A} \ t = \text{ta-der-strict } \mathcal{A} \ t

proof

{**fix** *q* **assume** *q \in | \text{ta-der } \mathcal{A} \ t* **then have** *q \in | \text{ta-der-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-der } \mathcal{A} \ t \subseteq | \text{ta-der-strict } \mathcal{A} \ t*

by *auto*

next

show *ta-der-strict } \mathcal{A} \ t \subseteq | \text{ta-der } \mathcal{A} \ t* **using** *ta-der-strict-sub-ta-der* .

qed

lemma *ta-der-to-ta-strict*:

assumes q \in | \text{ta-der } A \ C \langle \text{Var } p \rangle **and** *ground-ctxt C*

shows $\exists q'. (p = q' \vee (p, q') \in | \text{(eps } A)|^{+}) \wedge q \in | \text{ta-der-strict } A \ C \langle \text{Var } q' \rangle$

using *assms*

proof (*induct C arbitrary: q p*)

case (*More f ss C ts*)

from *More(2)* **obtain** *qs q'* **where**

r: TA-rule f qs q' \in | \text{rules } A \ \text{length } qs = \text{Suc } (\text{length } ss + \text{length } ts) \ q' = q \vee
 $(q', q) \in | \text{(eps } A)|^{+}$ **and**

rec: \forall i < \text{length } qs. qs \ ! \ i \in | \text{ta-der } A \ ((ss @ C \langle \text{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'') \in | \text{(eps } A)|^{+}) \wedge qs \ ! \ \text{length } ss \in | \text{ta-der-strict } A \ C \langle \text{Var}

```

q'\^
  by auto (metis length-map less-add-Suc1 nth-append-length)
  then have  $\forall i < \text{length } qs. qs ! i \in | \text{ta-der-strict } A ((ss @ C \langle \text{Var } q' \rangle \# ts) ! i)$ 
    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⟨t⟩))  $\longleftrightarrow$  root-ctxt C
  using assms by (cases C) auto

```

```

inductive-set Q-inf for  $\mathcal{A}$  where
  trans:  $(p, q) \in Q\text{-inf } \mathcal{A} \implies (q, r) \in Q\text{-inf } \mathcal{A} \implies (p, r) \in Q\text{-inf } \mathcal{A}$ 
| rule:  $(\text{None}, \text{Some } f) qs \rightarrow q \in | \text{rules } \mathcal{A} \implies i < \text{length } qs \implies (qs ! i, q) \in Q\text{-inf } \mathcal{A}$ 
| eps:  $(p, q) \in Q\text{-inf } \mathcal{A} \implies (q, r) \in | \text{eps } \mathcal{A} \implies (p, r) \in Q\text{-inf } \mathcal{A}$ 

```

```

abbreviation Q-inf-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  $p \in | \mathcal{Q} \mathcal{A} \ q \in | \mathcal{Q} \mathcal{A}$ 
  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 *:  $Q\text{-inf } \mathcal{A} \subseteq \text{fset } (\mathcal{Q} \mathcal{A} \mid \times \mid \mathcal{Q} \mathcal{A}) \ Q\text{-inf-e } \mathcal{A} \subseteq \text{fset } (\mathcal{Q} \mathcal{A})$ 
  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 $fQ\text{-inf} :: ('a, 'b \text{ option} \times 'c \text{ option}) \text{ ta} \Rightarrow ('a \times 'a) \text{ fset is } Q\text{-inf}$
by (*simp add: Q-inf-finite(1)*)
lift-definition $fQ\text{-inf-e} :: ('a, 'b \text{ option} \times 'c \text{ option}) \text{ ta} \Rightarrow 'a \text{ fset is } Q\text{-inf-e}$
using $Q\text{-inf-finite}(2)$.
end

lemma $Q\text{-inf-ta-eps-Q-inf}$:
assumes $(p, q) \in Q\text{-inf } \mathcal{A}$ **and** $(q, q') \in | \in | (\text{eps } \mathcal{A})^+ |$
shows $(p, q') \in Q\text{-inf } \mathcal{A}$ **using** $\text{assms}(2, 1)$
by (*induct rule: ftrancl-induct*) (*auto simp add: Q-inf.eps*)

lemma lhs-state-rule :
assumes $(p, q) \in Q\text{-inf } \mathcal{A}$
shows $\exists f \text{ qs } r. (\text{None}, \text{Some } f) \text{ qs} \rightarrow r \in | \in | \text{rules } \mathcal{A} \wedge p \in | \in | \text{fset-of-list qs}$
using assms **by** *induct* (*force intro: nth-mem*)⁺

lemma $Q\text{-inf-reach-state-rule}$:
assumes $(p, q) \in Q\text{-inf } \mathcal{A}$ **and** $Q \mathcal{A} \subseteq | \subseteq | \text{ta-reachable } \mathcal{A}$
shows $\exists \text{ ss } \text{ts } f \text{ C}. q \in | \in | \text{ta-der } \mathcal{A} (\text{More } (\text{None}, \text{Some } f) \text{ ss } \text{C } \text{ts}) (\text{Var } p) \wedge$
 $\text{ground-ctxt } (\text{More } (\text{None}, \text{Some } f) \text{ ss } \text{C } \text{ts})$
 $(\text{is } \exists \text{ ss } \text{ts } f \text{ C}. ?P \text{ ss } \text{ts } f \text{ C } q \text{ p})$
using assms
proof (*induct*)
case (*trans p q r*)
then obtain $f1 \text{ f2 } \text{ss1 } \text{ts1 } \text{ss2 } \text{ts2 } \text{C1 } \text{C2}$ **where**
 $\text{C}: ?P \text{ ss1 } \text{ts1 } f1 \text{ C1 } q \text{ p } ?P \text{ ss2 } \text{ts2 } f2 \text{ C2 } r \text{ q}$ **by** *blast*
then show *?case*
apply (*rule-tac x = ss2 in exI, rule-tac x = ts2 in exI, rule-tac x = f2 in exI,*
 $\text{rule-tac } x = \text{C2} \circ_c (\text{More } (\text{None}, \text{Some } f1) \text{ ss1 } \text{C1 } \text{ts1}) \text{ in exI}$)
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 $\forall i < \text{length } \text{qs}. \exists t. \text{qs} ! i \in | \in | \text{ta-der } \mathcal{A} \text{ t} \wedge \text{ground } t$
using $\text{rule}(1, 2) \text{ fset-mp}[OF \text{rule}(3), \text{of } \text{qs} ! i \text{ for } i]$
by *auto* (*meson fnth-mem rule-statesD(4) ta-reachableE*)
then obtain ts **where** $\text{wit}: \text{length } \text{ts} = \text{length } \text{qs}$
 $\forall i < \text{length } \text{qs}. \text{qs} ! i \in | \in | \text{ta-der } \mathcal{A} (\text{ts} ! i) \wedge \text{ground } (\text{ts} ! i)$
using $\text{Ex-list-of-length-P}[\text{of length } \text{qs } \lambda x \text{ i}. \text{qs} ! i \in | \in | \text{ta-der } \mathcal{A} \text{ x} \wedge \text{ground } x]$
by *blast*
{fix j assume $j < \text{length } \text{qs}$
then have $\text{qs} ! j \in | \in | \text{ta-der } \mathcal{A} ((\text{take } i \text{ ts} @ \text{Var } (\text{qs} ! i) \# \text{drop } (\text{Suc } i) \text{ ts}) ! j)$
using wit **by** (*cases j < i*) (*auto simp: min-def nth-append-Cons*)
then have $\forall i < \text{length } \text{qs}. \text{qs} ! i \in | \in | (\text{map } (\text{ta-der } \mathcal{A}) (\text{take } i \text{ ts} @ \text{Var } (\text{qs} ! i)$
 $\# \text{drop } (\text{Suc } i) \text{ ts})) ! i$
using wit $\text{rule}(2)$ **by** (*auto simp: nth-append-Cons*)

```

then have res:  $q \in | ta\text{-}der \mathcal{A} (Fun (None, Some f) (take i ts @ Var (qs ! i) \# drop (Suc i) ts))$ 
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)  $qs \rightarrow q' \in | rules \mathcal{A}$  and  $i < length qs$ 
shows  $(qs ! i, q') \in Q\text{-}inf \mathcal{A}$  using assms
by (intro rule) auto

```

```

lemma rule-target-eps-Q-inf:
assumes (None, Some f)  $qs \rightarrow q' \in | rules \mathcal{A} (q', q) \in | (eps \mathcal{A})|^{+}$ 
and  $i < length qs$ 
shows  $(qs ! i, q) \in Q\text{-}inf \mathcal{A}$ 
using assms(2, 1, 3) by (induct rule: ftrancl-induct) (auto intro: rule eps)

```

```

lemma step-in-Q-inf:
assumes  $q \in | ta\text{-}der\text{-}strict \mathcal{A} (map\text{-}funs\text{-}term (\lambda f. (None, Some f)) (Fun f (ss @ Var p \# ts)))$ 
shows  $(p, q) \in Q\text{-}inf \mathcal{A}$ 
using assms rule-target-eps-Q-inf[of f - -  $\mathcal{A}$  q] rule-target-Q-inf[of f - q  $\mathcal{A}$ ]
by (auto simp: comp-def nth-append-Cons split!: if-splits)

```

```

lemma ta-der-Q-inf:
assumes  $q \in | ta\text{-}der\text{-}strict \mathcal{A} (map\text{-}funs\text{-}term (\lambda f. (None, Some f)) (C \langle Var p \rangle))$ 
and  $C \neq Hole$ 
shows  $(p, q) \in Q\text{-}inf \mathcal{A}$  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  $q: q' \in | ta\text{-}der\text{-}strict \mathcal{A} (map\text{-}funs\text{-}term (\lambda f. (None, Some f)) C \langle Var p \rangle)$ 

```

```

    q |∈| ta-der-strict  $\mathcal{A}$  (map-funs-term ( $\lambda f. (None, Some f)$ ) (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  $\mathcal{A}$  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  $\mathcal{A}$  and  $\mathcal{Q} \mathcal{A} \sqsubseteq ta\text{-reachable } \mathcal{A}$ 
  shows infinite {t. q |∈| ta-der  $\mathcal{A}$  (term-of-gterm t) ∧ fst (groot-sym t) = None}
  proof -
    let ?P =  $\lambda u. ground\ u \wedge q \mid\in\ ta\text{-der } \mathcal{A}\ u \wedge fst (fst (the (root\ u))) = None$ 
    have groot[simp]:  $fst (fst (the (root (term-of-gterm\ t)))) = fst (groot-sym\ t)$  for
t by (cases t) auto
    have [simp]:  $C \neq \square \implies fst (fst (the (root\ C\langle t\rangle))) = fst (root-ctxt\ C)$  for C t
by (cases C) auto
    from assms(1) obtain p where cycle: (p, p) ∈ Q-inf  $\mathcal{A}$  (p, q) ∈ Q-inf  $\mathcal{A}$  by
auto
    from Q-inf-reach-state-rule[OF cycle(1) assms(2)] obtain C where
      ctxt:  $C \neq \square$  ground-ctxt C and reach:  $p \mid\in\ ta\text{-der } \mathcal{A}\ C\langle Var\ p\rangle$ 
    by blast
    obtain C2 where
      closing-ctxt:  $C2 \neq \square$  ground-ctxt C2  $fst (root-ctxt\ C2) = None$  and cl-reach:  $q \mid\in\ ta\text{-der } \mathcal{A}\ C2\langle Var\ p\rangle$ 
    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:  $p \mid\in\ ta\text{-der } \mathcal{A}\ t$  and gr-t: ground t
    by (meson Q-inf-states-ta-states(1) cycle(1) fsubsetD ta-reachableE)
    let ?terms =  $\lambda n. (C \hat{\ } Suc\ n)\langle t\rangle$  let ?S = {?terms n | n.  $p \mid\in\ ta\text{-der } \mathcal{A}\ (?terms\ n) \wedge ground\ (?terms\ n)$ }
    have ground (?terms n) for n using ctxt(2) gr-t by auto
    moreover have  $p \mid\in\ ta\text{-der } \mathcal{A}\ (?terms\ n)$  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 prod-automata-from-none-root-dec:
  assumes gta-lang Q A ⊆ {gpair s t | s t. funas-gterm s ⊆ F ∧ funas-gterm t ⊆ F}
  and q |∈ ta-der A (term-of-gterm t) and fst (groot-sym t) = None
  and Q A |⊆ ta-reachable A and q |∈ ta-productive Q A

```

shows $\exists u. t = \text{gterm-to-None-Some } u \wedge \text{funas-gterm } u \subseteq \mathcal{F}$
proof –
have $*$: $\text{gfun-at } t \ [] = \text{Some } (\text{groot-sym } t)$ **by** $(\text{cases } t)$ **auto**
from $\text{assms}(4, 5)$ **obtain** $C \ q_f$ **where** ctxt : $\text{ground-ctxt } C$ **and**
 fin : $q_f \ |\in| \ \text{ta-der } \mathcal{A} \ C \langle \text{Var } q \rangle \ q_f \ |\in| \ Q$
by $(\text{auto simp: ta-productive-def}'[OF \ \text{assms}(4)])$
then obtain $s \ v$ **where** gp : $\text{gpair } s \ v = (\text{gctxt-of-ctxt } C) \langle t \rangle_G$ **and**
 funas : $\text{funas-gterm } v \subseteq \mathcal{F}$
using $\text{assms}(1, 2)$ $\text{gta-langI}[OF \ \text{fin}(2), \text{of } \mathcal{A} \ (\text{gctxt-of-ctxt } C) \langle t \rangle_G]$
by $(\text{auto simp: ta-der-ctxt ground-gctxt-of-ctxt-apply-gterm})$
from gp **have** mem : $\text{hole-pos } C \in \text{gposs } s \cup \text{gposs } v$
by auto $(\text{metis Un-iff ctxt ghole-pos-in-apply gposs-of-gpair ground-hole-pos-to-ghole})$
from this **have** $\text{hole-pos } C \notin \text{gposs } s$ **using** $\text{assms}(3)$
using $\text{arg-cong}[OF \ \text{gp}, \text{of } \lambda t. \text{gfun-at } t \ (\text{hole-pos } C)]$
using $\text{ground-hole-pos-to-ghole}[OF \ \text{ctxt}]$
using $\text{gfun-at-after-hole-pos}[\text{of } \text{gctxt-of-ctxt } C]$
by $(\text{auto simp: gfun-at-gpair } * \ \text{split: if-splits})$
 $(\text{metis fstI gfun-at-None-ngposs-iff})+$
from $\text{subst-at-gpair-nt-poss-None-Some}[OF \ - \ \text{this}, \text{of } v]$ **this**
have $t = \text{gterm-to-None-Some } (\text{gsubt-at } v \ (\text{hole-pos } C)) \wedge \text{funas-gterm } (\text{gsubt-at } v \ (\text{hole-pos } C)) \subseteq \mathcal{F}$
using $\text{funas mem funas-gterm-gsubt-at-subseteq}$
by $(\text{auto simp: gp intro!: exI}[\text{of } - \ \text{gsubt-at } v \ (\text{hole-pos } C)])$
 $(\text{metis ctxt ground-hole-pos-to-ghole gsubt-at-gctxt-apply-ghole})$
then show $?thesis$ **by blast**
qed

lemma *infinite-set-dec-infinite*:

assumes *infinite* S **and** $\bigwedge s. s \in S \implies \exists t. f \ t = s \wedge P \ t$
shows *infinite* $\{t \mid t \ s. s \in S \wedge f \ t = s \wedge P \ t\}$ **(is infinite ?T)**
proof (rule ccontr)

assume $\text{ass}: \neg \text{infinite } ?T$
have $S \subseteq f \ ' \ \{x . P \ x\}$ **using** $\text{assms}(2)$ **by auto**
then show *False* **using** ass $\text{assms}(1)$
by $(\text{auto simp: subset-image-iff})$
 $(\text{metis } (\text{mono-tags}, \text{lifting}) \ \text{Ball-Collect finite-imageI image-eqI infinite-super})$

qed

lemma *Q-inf-exec-impl-Q-inf*:

assumes $\text{gta-lang } Q \ \mathcal{A} \subseteq \{\text{gpair } s \ t \mid s \ t. \text{funas-gterm } s \subseteq \text{fset } \mathcal{F} \wedge \text{funas-gterm } t \subseteq \text{fset } \mathcal{F}\}$

and $Q \ \mathcal{A} \ |\subseteq| \ \text{ta-reachable } \mathcal{A}$ **and** $Q \ \mathcal{A} \ |\subseteq| \ \text{ta-productive } Q \ \mathcal{A}$

and $q \in Q\text{-inf-e } \mathcal{A}$

shows $q \ |\in| \ Q\text{-infy } \mathcal{A} \ \mathcal{F}$

proof –

let $?S = \{t. q \ |\in| \ \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } t) \wedge \text{fst } (\text{groot-sym } t) = \text{None}\}$

let $?P = \lambda t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q \ |\in| \ \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } (\text{gterm-to-None-Some } t))$

let $?F = (\lambda(f, n). ((\text{None}, \text{Some } f), n)) \ |\ ' \ \mathcal{F}$

from $Q\text{-inf-e-infinite-terms-res}[OF\ assms(4, 2)]$ **have** $inf: infinite\ ?S$ **by** *auto*
{fix t **assume** $t \in ?S$
then **have** $\exists u. t = gterm\text{-to-None-Some}\ u \wedge funas\text{-gterm}\ u \subseteq fset\ \mathcal{F}$
using $prod\text{-automata-from-none-root-dec}[OF\ assms(1)]\ assms(2, 3)$
using *fin-mono* **by** *fastforce*
then **show** $?thesis$ **using** $infinite\text{-set-dec-infinite}[OF\ inf, of\ gterm\text{-to-None-Some}\ ?P]$
by (*auto simp: Q-inf-fty-fmember*) *blast*
qed

lemma $Q\text{-inf-impl-Q-inf-exec}$:

assumes $q \in | Q\text{-inf-fty}\ \mathcal{A}\ \mathcal{F}$
shows $q \in Q\text{-inf-e}\ \mathcal{A}$

proof –

let $?t\text{-of-g} = \lambda t. term\text{-of-gterm}\ t :: ('b\ option \times 'b\ option, 'a)\ term$
let $?t\text{-og-g2} = \lambda t. term\text{-of-gterm}\ t :: ('b, 'a)\ term$
let $?inf = (?t\text{-og-g2} :: 'b\ gterm \Rightarrow ('b, 'a)\ term) \text{ ‘ } \{t \mid t. funas\text{-gterm}\ t \subseteq fset\ \mathcal{F} \wedge q \in | ta\text{-der}\ \mathcal{A}\ (?t\text{-of-g}\ (gterm\text{-to-None-Some}\ t))\}$
obtain n **where** $card\text{-st: } fcard\ (Q\ \mathcal{A}) < n$ **by** *blast*
from $assms(1)$ **have** $infinite\ \{t \mid t. funas\text{-gterm}\ t \subseteq fset\ \mathcal{F} \wedge q \in | ta\text{-der}\ \mathcal{A}\ (?t\text{-of-g}\ (gterm\text{-to-None-Some}\ t))\}$
unfolding $Q\text{-inf-fty-def}$ **by** *blast*
from $infinite\text{-inj-image-infinite}[OF\ this, of\ ?t\text{-og-g2}]$ **have** $inf: infinite\ ?inf$ **using** *inj-term-of-gterm* **by** *blast*
{fix s **assume** $s \in ?inf$ **then** **have** $ground\ s\ funas\text{-term}\ s \subseteq fset\ \mathcal{F}$ **by** (*auto simp: funas-term-of-gterm-conv subsetD*)
from $infinte\text{-no-depth-limit}[OF\ inf, of\ fset\ \mathcal{F}]$ **this** **obtain** u **where**
 $funas: funas\text{-gterm}\ u \subseteq fset\ \mathcal{F}$ **and** $card\text{-d: } n < depth\ (?t\text{-og-g2}\ u)$ **and** $reach: q \in | ta\text{-der}\ \mathcal{A}\ (?t\text{-of-g}\ (gterm\text{-to-None-Some}\ u))$
by *auto blast*
have $depth\ (?t\text{-og-g2}\ u) = depth\ (?t\text{-of-g}\ (gterm\text{-to-None-Some}\ u))$
proof (*induct u*)
case ($GFun\ f\ ts$) **then** **show** $?case$
by (*auto simp: comp-def intro: nth-args-depth-eqI*)
qed

from $this\ pigeonhole\text{-tree-automata}[OF\ \text{-}\ reach]\ card\text{-st}\ card\text{-d}$ **obtain** $C2\ C\ s\ v$
 p **where**

$ctxt: C2 \neq \square\ C\langle s \rangle = term\text{-of-gterm}\ (gterm\text{-to-None-Some}\ u)\ C2\langle v \rangle = s$ **and**
 $loop: p \in | ta\text{-der}\ \mathcal{A}\ v \wedge p \in | ta\text{-der}\ \mathcal{A}\ C2\langle Var\ p \rangle \wedge q \in | ta\text{-der}\ \mathcal{A}\ C\langle Var\ p \rangle$
by *auto*

from $ctxt$ **have** $gr: ground\ ctxt\ C2\ ground\ ctxt\ C$ **by** *auto* (*metis ground-ctxt-apply ground-term-of-gterm*) $+$

from $ta\text{-der-to-ta-strict}[OF\ \text{-}\ gr(1)]$ *loop* **obtain** q' **where**

$to\text{-strict: } (p = q' \vee (p, q') \in | (eps\ \mathcal{A})|^{+}) \wedge p \in | ta\text{-der-strict}\ \mathcal{A}\ C2\langle Var\ q' \rangle$

by *fastforce*

have $*$: $\exists C. C2 = map\text{-funs-ctxt}\ lift\text{-None-Some}\ C \wedge C \neq \square$ **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) ∈ Q-inf A using to-strict ta-der-Q-inf[of p A - q'] ctat
    by auto
  then have cycle: (q', q') ∈ Q-inf A using to-strict by (auto intro: Q-inf-ta-eps-Q-inf)
  show ?thesis
  proof (cases C = □)
    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'' ∨ (p, q'') |∈| (eps A)|+ q |∈| ta-der-strict A
          C⟨Var q'⟩
          using ta-der-to-ta-strict[OF conjunct2[OF conjunct2[OF loop]] gr(2)] by auto
          then have (q'', q) ∈ Q-inf A using ta-der-Q-inf[of q A - q'] ctat False
          by auto (metis (mono-tags, lifting) map-ctat.simps(1) map-funs-term-ctat-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 A ⊆ {gpair s t | s t. funas-gterm s ⊆ fset F ∧ funas-gterm
t ⊆ fset F}
  and Q A |⊆| ta-reachable A and Q A |⊆| ta-productive Q A
  shows Q-infty A F = fQ-inf-e 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 L A ⊆ {gpair s t | s t. funas-gterm s ⊆ fset F ∧ funas-gterm t ⊆ fset
F}
  and Qr A |⊆| ta-reachable (ta A) and Qr A |⊆| ta-productive (fin A) (ta A)
  shows L (Inf-reg-impl A) = L (Inf-reg A (Q-infty (ta A) F))
  using Q-infty-fQ-inf-e-conv[of fin A ta A F] assms[unfolded 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 $\Delta \Delta_E \equiv TA$ (*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* \Rightarrow (*'q, 'f*) *ta* \Rightarrow (*'q* \times *'q*) *horn set* **where**

Δ_ε -rules *A B* =
 $\{zip\ ps\ qs \rightarrow_h (p, q) \mid f\ ps\ p\ qs\ q. f\ ps \rightarrow p \mid \in \mid rules\ A \wedge f\ qs \rightarrow q \mid \in \mid rules\ B$
 $\wedge\ length\ ps = length\ qs\} \cup$
 $\{[(p, q)] \rightarrow_h (p', q) \mid p\ p'\ q. (p, p') \mid \in \mid eps\ A\} \cup$
 $\{[(p, q)] \rightarrow_h (p, q') \mid p\ q\ q'. (q, q') \mid \in \mid eps\ B\}$

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 = $\{(p, q) \mid f\ p\ q. f\ [] \rightarrow p \mid \in \mid rules\ A \wedge f\ [] \rightarrow q \mid \in \mid rules\ B\}$

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* = $\{(p, q) \mid f\ ps\ p\ qs\ q. f\ ps \rightarrow p \mid \in \mid rules\ A \wedge f\ qs \rightarrow q \mid \in \mid rules\ B \wedge$
 $length\ ps = length\ qs \wedge$

$(fst\ pq, snd\ pq) \in set\ (zip\ ps\ qs) \wedge set\ (zip\ ps\ qs) \subseteq insert\ pq\ X\} \cup$

$\{(p', snd\ pq) \mid p\ p'. (p, p') \mid \in \mid eps\ A \wedge p = fst\ pq\} \cup$

$\{(fst\ pq, q') \mid q\ q'. (q, q') \mid \in \mid eps\ B \wedge q = snd\ pq\}$

unfolding Δ_ε -rules-def *horn-infer1-union*

apply (*intro arg-cong2*[*of* $---$ (\cup)])

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 (Δ_ε -set-cong *f ps p qs q*) **show** *?case*

apply (*intro infer*[*of zip ps qs (p, q)*])

subgoal **using** Δ_ε -set-cong(1-3) **by** (*auto simp: \Delta_\varepsilon*-rules-def)

subgoal **using** Δ_ε -set-cong(3,5) **by** (*auto simp: zip-nth-conv*)

done

next

case (Δ_ε -set-eps1 *p p' q*) **then show** *?case*

```

    by (intro infer[of [(p, q)] (p', q)]) (auto simp:  $\Delta_\epsilon$ -rules-def)
  next
    case ( $\Delta_\epsilon$ -set-eps2 q q' p) then show ?case
      by (intro infer[of [(p, q)] (p, q')]) (auto simp:  $\Delta_\epsilon$ -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  $\Delta_\epsilon$ -set-cong[of - map fst as fst a A map snd as snd a B]
               $\Delta_\epsilon$ -set-eps1[of - fst a A snd a B]  $\Delta_\epsilon$ -set-eps2[of - snd a B fst a A]
        by (auto simp:  $\Delta_\epsilon$ -rules-def)
      qed
    qed
  end

```

11 Computing the epsilon transitions for the transitive closure of GTT's

definition Δ -trancl-rules :: $(\text{'q}, \text{'f}) \text{ ta} \Rightarrow (\text{'q}, \text{'f}) \text{ ta} \Rightarrow (\text{'q} \times \text{'q}) \text{ horn set}$ where
 Δ -trancl-rules A B =
 Δ_ϵ -rules A B $\cup \{[(p, q), (q, r)] \rightarrow_h (p, r) \mid p \ q \ r. \text{ True}\}$

locale Δ -trancl-horn =
 fixes A :: $(\text{'q}, \text{'f}) \text{ ta}$ and B :: $(\text{'q}, \text{'f}) \text{ ta}$
 begin

sublocale horn Δ -trancl-rules A B .

lemma Δ -trancl-infer0:
 infer0 = horn.infer0 (Δ_ϵ -rules A B)
 by (auto simp: Δ_ϵ -rules-def Δ -trancl-rules-def horn.infer0-def)

lemma Δ -trancl-infer1:
 infer1 pq X = horn.infer1 (Δ_ϵ -rules A B) pq X \cup
 $\{(r, \text{snd } pq) \mid r \ p'. (r, p') \in X \wedge p' = \text{fst } pq\} \cup$
 $\{(\text{fst } pq, r) \mid q' \ r. (q', r) \in (\text{insert } pq \ X) \wedge q' = \text{snd } pq\}$
unfolding Δ -trancl-rules-def horn-infer1-union Un-assoc
apply (intro arg-cong2[of - - - (\cup)] 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 ( $\Delta$ -set-cong f ps p qs q) show ?case
    apply (intro infer[of zip ps qs (p, q)])
  subgoal using  $\Delta$ -set-cong(1-3) by (auto simp:  $\Delta$ -trancl-rules-def  $\Delta_\varepsilon$ -rules-def)
  subgoal using  $\Delta$ -set-cong(3,5) by (auto simp: zip-nth-conv)
  done
next
  case ( $\Delta$ -set-eps1 p p' q) then show ?case
    by (intro infer[of [(p, q)] (p', q)]) (auto simp:  $\Delta$ -trancl-rules-def  $\Delta_\varepsilon$ -rules-def)
next
  case ( $\Delta$ -set-eps2 q q' p) then show ?case
    by (intro infer[of [(p, q)] (p, q')]) (auto simp:  $\Delta$ -trancl-rules-def  $\Delta_\varepsilon$ -rules-def)
next
  case ( $\Delta$ -set-trans p q r) then show ?case
    by (intro infer[of [(p,q), (q,r)] (p, r)]) (auto simp:  $\Delta$ -trancl-rules-def  $\Delta_\varepsilon$ -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  $\Delta$ -set-cong[of - map fst as fst a A map snd as snd a B]
         $\Delta$ -set-eps1[of - fst a A snd a B]  $\Delta$ -set-eps2[of - snd a B fst a A]
         $\Delta$ -set-trans[of fst a snd (hd as) A B snd a]
      by (auto simp:  $\Delta$ -trancl-rules-def  $\Delta_\varepsilon$ -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**

$$\Delta\text{-Atr-rules } Q A B = \{[] \rightarrow_h (p, q) \mid p q. (p, q) \in Q\} \cup \{[(p, q), (r, v)] \rightarrow_h (p, v) \mid p q r v. (q, r) \in \Delta_\varepsilon B A\}$$

locale Δ -Atr-horn =

fixes Q :: ('q × '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:
 $infer1\ pq\ X = \{(p, snd\ pq) \mid p\ q. (p, q) \in X \wedge (q, fst\ pq) \mid\in\ \Delta_\varepsilon\ B\ A\} \cup$
 $\{(fst\ pq, v) \mid q\ r\ v. (snd\ pq, r) \mid\in\ \Delta_\varepsilon\ B\ A \wedge (r, v) \in X\} \cup$
 $\{(fst\ pq, snd\ pq) \mid q. (snd\ pq, fst\ pq) \mid\in\ \Delta_\varepsilon\ B\ A\}$
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 \times 'g option) ta \Rightarrow ('q \times 'q) horn set **where**
 Q -inf-rules $A =$
 $\{\ [] \rightarrow_h (ps\ !\ i, p) \mid f\ ps\ p\ i. (None, Some\ f)\ ps \rightarrow p \mid\in\ rules\ A \wedge i < length\ ps\}$
 \cup
 $\{[(p, q)] \rightarrow_h (p, r) \mid p\ q\ r. (q, r) \mid\in\ eps\ A\} \cup$
 $\{[(p, q), (q, r)] \rightarrow_h (p, r) \mid p\ q\ r. True\}$

locale Q -horn =
fixes $A :: ('q, 'f\ option \times 'g\ option)\ ta$
begin

sublocale *horn Q-inf-rules A* .

lemma *Q-infer0*:

infer0 = $\{(ps ! i, p) \mid f ps p i. (None, Some f) ps \rightarrow p \mid \in \text{rules } A \wedge i < \text{length } ps\}$

unfolding *horn.infer0-def Q-inf-rules-def* **by** *auto*

lemma *Q-infer1*:

infer1 *pq X* = $\{(fst pq, r) \mid q r. (q, r) \mid \in \text{eps } A \wedge q = snd pq\} \cup$
 $\{(r, snd pq) \mid r p'. (r, p') \in X \wedge p' = fst pq\} \cup$
 $\{(fst pq, r) \mid q' r. (q', r) \in (\text{insert } pq X) \wedge q' = snd pq\}$

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**

```

 $\Delta_\varepsilon$ -infer0-cont  $\Delta_A \Delta_B =$ 
  (let arules = filter ( $\lambda r$ . r-lhs-states  $r = []$ ) (sorted-list-of-fset  $\Delta_A$ ) in
   let brules = filter ( $\lambda r$ . r-lhs-states  $r = []$ ) (sorted-list-of-fset  $\Delta_B$ ) in
   (map (map-prod r-rhs r-rhs) (filter ( $\lambda(ra, rb)$ . r-root  $ra = r$ -root  $rb$ ) (List.product
arules brules))))

```

definition Δ_ε -infer1-cont **where**

```

 $\Delta_\varepsilon$ -infer1-cont  $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} =$ 
  (let (arules, aeps) = (sorted-list-of-fset  $\Delta_A$ , sorted-list-of-fset  $\Delta_{A\varepsilon}$ ) in
   let (brules, beps) = (sorted-list-of-fset  $\Delta_B$ , sorted-list-of-fset  $\Delta_{B\varepsilon}$ ) in
   let prules = List.product arules brules in
   ( $\lambda pq$  bs.
    map (map-prod r-rhs r-rhs) (filter ( $\lambda(ra, rb)$ . case (ra, rb) of (TA-rule f ps p,
TA-rule g qs q)  $\Rightarrow$ 
      f = g  $\wedge$  length ps = length qs  $\wedge$  (fst pq, snd pq)  $\in$  set (zip ps qs)  $\wedge$ 
      set (zip ps qs)  $\subseteq$  insert (fst pq, snd pq) (fset bs)) prules) @
    map ( $\lambda(p, p')$ . (p', snd pq)) (filter ( $\lambda(p, p')$   $\Rightarrow p =$  fst pq) aeps) @
    map ( $\lambda(q, q')$ . (fst pq, q')) (filter ( $\lambda(q, q')$   $\Rightarrow q =$  snd pq) beps)))

```

locale Δ_ε -fset =

```

  fixes  $\Delta_A :: ('q :: linorder, 'f :: linorder)$  ta-rule fset and  $\Delta_{A\varepsilon} :: ('q \times 'q)$  fset
  and  $\Delta_B :: ('q, 'f)$  ta-rule fset and  $\Delta_{B\varepsilon} :: ('q \times 'q)$  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 $A B \Delta_\varepsilon$ -infer0-cont $\Delta_A \Delta_B \Delta_\varepsilon$ -infer1-cont Δ_A
 $\Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$

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**

```

 $\Delta_\varepsilon$ -impl  $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} = \text{horn-fset-impl.saturate-impl } (\Delta_\varepsilon\text{-infer0-cont } \Delta_A
\Delta_B) (\Delta_\varepsilon\text{-infer1-cont } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon})$ 
```

lemma Δ_ε -impl-sound:

```

assumes  $\Delta_\varepsilon$ -impl  $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} = \text{Some } xs$ 
shows  $xs = \Delta_\varepsilon (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon})$ 
using  $\Delta_\varepsilon$ -fset.saturate-impl-sound[OF assms[unfolded  $\Delta_\varepsilon$ -impl-def]]
unfolding  $\Delta_\varepsilon$ -horn. $\Delta_\varepsilon$ -sound[symmetric]
by (auto simp flip:  $\Delta_\varepsilon$ .rep-eq)

```

lemma Δ_ε -impl-complete:

```

fixes  $\Delta_A :: ('q :: \text{linorder}, 'f :: \text{linorder}) \text{ ta-rule fset}$  and  $\Delta_B :: ('q, 'f) \text{ ta-rule}$ 
fset
  and  $\Delta_{\varepsilon A} :: ('q \times 'q) \text{ fset}$  and  $\Delta_{\varepsilon B} :: ('q \times 'q) \text{ fset}$ 
shows  $\Delta_\varepsilon$ -impl  $\Delta_A \Delta_{\varepsilon A} \Delta_B \Delta_{\varepsilon B} \neq \text{None}$  unfolding  $\Delta_\varepsilon$ -impl-def
by (intro  $\Delta_\varepsilon$ -fset.saturate-impl-complete)
  (auto simp flip:  $\Delta_\varepsilon$ -horn. $\Delta_\varepsilon$ -sound)

```

lemma Δ_ε -impl [code]:

```

 $\Delta_\varepsilon (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon}) = \text{the } (\Delta_\varepsilon\text{-impl } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon})$ 
using  $\Delta_\varepsilon$ -impl-complete[of  $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$ ]  $\Delta_\varepsilon$ -impl-sound[of  $\Delta_A \Delta_{A\varepsilon} \Delta_B$ 
 $\Delta_{B\varepsilon}$ ]
by force

```

15 Computing the epsilon transitions for the transitive closure of GTT's

definition Δ -trancl-infer0 **where**

$$\Delta\text{-trancl-infer0 } \Delta_A \Delta_B = \Delta_\varepsilon\text{-infer0-cont } \Delta_A \Delta_B$$

definition Δ -trancl-infer1 :: ('q :: linorder , 'f :: linorder) ta-rule fset \Rightarrow ('q \times 'q) fset \Rightarrow ('q, 'f) ta-rule fset \Rightarrow ('q \times 'q) fset

\Rightarrow 'q \times 'q \Rightarrow ('q \times 'q) fset \Rightarrow ('q \times 'q) list **where**

$$\Delta\text{-trancl-infer1 } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} pq bs =$$

$$\Delta_\varepsilon\text{-infer1-cont } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} pq bs @$$

sorted-list-of-fset (

$$(\lambda(r, p'). (r, \text{snd } pq)) \mid \mid (\text{ffilter } (\lambda(r, p') \Rightarrow p' = \text{fst } pq) bs) \mid \cup \mid$$

$$(\lambda(q', r). (\text{fst } pq, r)) \mid \mid (\text{ffilter } (\lambda(q', r) \Rightarrow q' = \text{snd } pq) (\text{finsert } pq bs)))$$

locale Δ -trancl-list =

fixes Δ_A :: ('q :: linorder, 'f :: linorder) ta-rule fset **and** $\Delta_{A\varepsilon}$:: ('q \times 'q) fset

and Δ_B :: ('q, 'f) ta-rule fset **and** $\Delta_{B\varepsilon}$:: ('q \times '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 -trancl-horn A B .

sublocale l: horn-fset Δ -trancl-rules A B

$$\Delta\text{-trancl-infer0 } \Delta_A \Delta_B \Delta\text{-trancl-infer1 } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$$

apply (unfold-locales)

unfolding Δ -trancl-rules-def horn-infer0-union horn-infer1-union

$$\Delta\text{-trancl-infer0-def } \Delta\text{-trancl-infer1-def } \Delta_\varepsilon\text{-fset.infer set-append}$$

by (auto simp flip: ex-simps(1) simp: horn.infer0-def horn.infer1-def intro!: arg-cong2[of - - - (U)])

lemmas saturate-impl-sound = l.saturate-impl-sound

lemmas saturate-impl-complete = l.saturate-impl-complete

end

definition Δ -trancl-impl $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} =$

$$\text{horn-fset-impl.saturate-impl } (\Delta\text{-trancl-infer0 } \Delta_A \Delta_B) (\Delta\text{-trancl-infer1 } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon})$$

lemma Δ -trancl-impl-sound:

assumes Δ -trancl-impl $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} = \text{Some } xs$

shows $xs = \Delta\text{-trancl } (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon})$

using Δ -trancl-list.saturate-impl-sound[OF assms[unfolding 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 $\Delta_A :: ('q :: \text{linorder}, 'f :: \text{linorder}) \text{ ta-rule fset}$ **and** $\Delta_B :: ('q, 'f) \text{ ta-rule fset}$

and $\Delta_{A\varepsilon} :: ('q \times 'q) \text{ fset}$ **and** $\Delta_{B\varepsilon} :: ('q \times 'q) \text{ fset}$

shows Δ -trancl-impl $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} \neq \text{None}$

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 ($TA \Delta_A \Delta_{A\varepsilon}$) ($TA \Delta_B \Delta_{B\varepsilon}$) = (*the* (Δ -trancl-impl $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$))

using Δ -trancl-impl-complete[*of* $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$]

using Δ -trancl-impl-sound[*of* $\Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$]

by force

16 Computing the epsilon transitions for the transitive closure of pair automata

definition Δ -Atr-infer1-cont $:: ('q :: \text{linorder} \times 'q) \text{ fset} \Rightarrow ('q, 'f :: \text{linorder}) \text{ ta-rule fset} \Rightarrow ('q \times 'q) \text{ fset} \Rightarrow$

$('q, 'f) \text{ ta-rule fset} \Rightarrow ('q \times 'q) \text{ fset} \Rightarrow 'q \times 'q \Rightarrow ('q \times 'q) \text{ fset} \Rightarrow ('q \times 'q) \text{ list}$

where

Δ -Atr-infer1-cont $Q \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} =$

(*let* $G = \text{sorted-list-of-fset}$ (*the* (Δ_ε -impl $\Delta_B \Delta_{B\varepsilon} \Delta_A \Delta_{A\varepsilon}$)) *in*

($\lambda pq \text{ bs.}$

(*let* $bs\text{-list} = \text{sorted-list-of-fset}$ bs *in*

pq) (*List.product* $bs\text{-list}$ G) @

pq) (*List.product* G $bs\text{-list}$) @

pq) (*List.product* ($\lambda (p, q). (fst pq, snd pq)$) (*filter* ($\lambda (p, q). snd pq = p \wedge fst pq = q$) G))))

locale Δ -Atr-fset =

fixes $Q :: ('q :: \text{linorder} \times 'q) \text{ fset}$ **and** $\Delta_A :: ('q, 'f :: \text{linorder}) \text{ ta-rule fset}$ **and** $\Delta_{A\varepsilon} :: ('q \times 'q) \text{ fset}$

and $\Delta_B :: ('q, 'f) \text{ ta-rule fset}$ **and** $\Delta_{B\varepsilon} :: ('q \times 'q) \text{ fset}$

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 $\{(p, snd pq) \mid p q. (p, q) \in (\text{fset } bs) \wedge (q, fst pq) \in \Delta_\varepsilon B A\} \cup$

```

    {(fst pq, v) | q r v. (snd pq, r) |∈| Δε B A ∧ (r, v) ∈ (fset bs)} ∪
    {(fst pq, snd pq) | q . (snd pq, fst pq) |∈| Δε B A} = set (Δ-Atr-infer1-cont Q
ΔA ΔAε ΔB ΔBε 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: Δε-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 ΔA ΔAε ΔB ΔBε
  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 ΔA ΔAε ΔB ΔBε =
  horn-fset-impl.saturate-impl (sorted-list-of-fset Q) (Δ-Atr-infer1-cont Q ΔA ΔAε
ΔB ΔBε)

```

```

lemma Δ-Atr-impl-sound:
  assumes Δ-Atr-impl Q ΔA ΔAε ΔB ΔBε = Some xs
  shows xs = Δ-Atrans Q (TA ΔA ΔAε) (TA ΔB ΔBε)
  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 ΔA ΔAε ΔB ΔBε ≠ 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 ΔA ΔAε) (TA ΔB ΔBε) = (the (Δ-Atr-impl Q ΔA ΔAε ΔB
ΔBε))
  using Δ-Atr-impl-complete[of Q ΔA ΔAε ΔB ΔBε] Δ-Atr-impl-sound[of Q ΔA
ΔAε ΔB ΔBε]
  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 × 'g) 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 × 'g) fset ⇒ 'q × 'g ⇒ ('q × 'g) fset ⇒ ('q × 'g) 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 × 'g) fset

begin

abbreviation *A* where $A \equiv TA \ \Delta \ \Delta\varepsilon$

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*:
 $Q\text{-impl } \Delta \Delta\varepsilon = \text{Some } xs \implies \text{fset } xs = Q\text{-inf } (TA \Delta \Delta\varepsilon)$
using *Q-fset.saturate-impl-sound unfolding Q-impl-def Q-horn.Q-sound* .

lemma *Q-impl-complete*:
 $Q\text{-impl } \Delta \Delta\varepsilon \neq \text{None}$
proof –
let $?A = TA \Delta \Delta\varepsilon$
have $*$: $Q\text{-inf } ?A \subseteq \text{fset } (\mathcal{Q} ?A \mid \times \mid \mathcal{Q} ?A)$
by (*auto simp add: Q-inf-states-ta-states(1, 2) subrelI*)
have *finite* ($Q\text{-inf } ?A$)
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* $\Delta \Delta\varepsilon = (\text{let } Q = \text{the } (Q\text{-impl } \Delta \Delta\varepsilon) \text{ in}$
 $\text{snd } \mid \uparrow \mid ((\text{ffilter } (\lambda (p, q). p = q) Q) \mid O \mid Q))$

lemma *Q-infinity-impl-fmember*:
 $q \mid \in \mid Q\text{-infinity-impl } \Delta \Delta\varepsilon \iff (\exists p. (p, p) \mid \in \mid \text{the } (Q\text{-impl } \Delta \Delta\varepsilon) \wedge$
 $(p, q) \mid \in \mid \text{the } (Q\text{-impl } \Delta \Delta\varepsilon))$
unfolding *Q-infinity-impl-def*
by (*auto simp: Let-def image-iff Bex-def*) *fastforce*

lemma *loop-sound-correct* [*simp*]:
 $\text{fset } (Q\text{-infinity-impl } \Delta \Delta\varepsilon) = Q\text{-inf-e } (TA \Delta \Delta\varepsilon)$
proof –
obtain Q **where** [*simp*]: $Q\text{-impl } \Delta \Delta\varepsilon = \text{Some } Q$ **using** *Q-impl-complete[of $\Delta \Delta\varepsilon$]*
by *blast*
have $\text{fset } Q = (Q\text{-inf } (TA \Delta \Delta\varepsilon))$
using *Q-impl-sound[of $\Delta \Delta\varepsilon$]*
by (*auto simp: fset-of-list.rep-eq*)
then show *?thesis*
by (*force simp: Q-infinity-impl-fmember Let-def fset-of-list-elim fset-of-list.rep-eq*)
qed

lemma *fQ-inf-e-code* [*code*]:
 $fQ\text{-inf-e } (TA \Delta \Delta\varepsilon) = Q\text{-infinity-impl } \Delta \Delta\varepsilon$
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.