

# 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Preliminaries</b>	<b>4</b>
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 . . . . .	7
2.5	Misc . . . . .	7
2.6	Ground constructions . . . . .	11

---

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

2.6.1	Ground terms . . . . .	11
2.6.2	Tree domains . . . . .	15
2.6.3	Ground context . . . . .	29
2.6.4	Multihole context closure . . . . .	34
2.6.5	Signature closed property . . . . .	35
2.6.6	Transitive closure preserves <i>all-ctxt-closed-gterm</i> . . . . .	35
<b>3</b>	<b>Tree automaton</b>	<b>40</b>
3.1	Tree automaton definition and functionality . . . . .	40
3.1.1	Reachability of a term induced by a tree automaton . . . . .	40
3.1.2	Language acceptance . . . . .	41
3.1.3	Trimming . . . . .	41
3.1.4	Mapping over tree automata . . . . .	42
3.1.5	Product construction (language intersection) . . . . .	43
3.1.6	Union construction (language union) . . . . .	43
3.1.7	Epsilon free and tree automaton accepting empty language . . . . .	43
3.1.8	Relabeling tree automaton states to natural numbers . . . . .	44
3.2	Powerset Construction for Tree Automata . . . . .	69
3.3	Complement closure of regular languages . . . . .	72
3.4	Pumping lemma . . . . .	74
3.5	Myhill Nerode characterization for regular tree languages . . . . .	78
<b>4</b>	<b>Ground Tree Transducers (GTT)</b>	<b>79</b>
4.1	(A)GTT reachable states . . . . .	82
4.2	(A)GTT productive states . . . . .	82
4.3	(A)GTT trimming . . . . .	83
4.4	root-cleanliness . . . . .	83
4.5	Relabeling . . . . .	83
4.6	epsilon free GTTs . . . . .	84
4.7	GTT closure under composition . . . . .	84
4.8	GTT closure under transitivity . . . . .	88
4.9	Pair automaton and anchored GTTs . . . . .	90
4.10	Anchored gtt composition . . . . .	95
4.11	Anchored gtt transitivity . . . . .	95
4.12	Anchored gtt intersection . . . . .	95
4.13	Anchored gtt trimming . . . . .	96
<b>5</b>	<b>Regular relations</b>	<b>96</b>
5.1	Encoding pairs of terms . . . . .	96
5.2	Decoding of pairs . . . . .	98
5.3	Contexts to gpair . . . . .	98
5.4	Encoding of lists of terms . . . . .	100
5.5	RRn relations . . . . .	101

5.6	Nullary automata . . . . .	102
5.7	Pairing RR1 languages . . . . .	102
5.8	Collapsing . . . . .	105
5.9	Cylindrification . . . . .	108
5.10	Projection . . . . .	108
5.11	Permutation . . . . .	109
5.12	Intersection . . . . .	109
5.13	Difference . . . . .	110
5.14	All terms over a signature . . . . .	110
5.15	RR2 composition . . . . .	111
<b>6</b>	<b>Computing state derivation</b>	<b>116</b>
<b>7</b>	<b>Computing the restriction of tree automata to state set</b>	<b>116</b>
<b>8</b>	<b>Computing the epsilon transition for the product automaton</b>	<b>117</b>
<b>9</b>	<b>Computing reachability</b>	<b>117</b>
9.1	Horn setup for reachable states . . . . .	117
9.2	Computing productivity . . . . .	118
9.2.1	Horn setup for productive states . . . . .	118
9.3	Horn setup for power set construction states . . . . .	119
9.4	Setup for the list implementation of reachable states . . . . .	124
9.5	Setup for list implementation of productive states . . . . .	125
9.6	Setup for the implementation of power set construction states	126
<b>10</b>	<b>Computing the epsilon transitions for the composition of GTT's</b>	<b>133</b>
<b>11</b>	<b>Computing the epsilon transitions for the transitive closure of GTT's</b>	<b>134</b>
<b>12</b>	<b>Computing the epsilon transitions for the transitive closure of pair automata</b>	<b>135</b>
<b>13</b>	<b>Computing the Q infinity set for the infinity predicate automaton</b>	<b>135</b>
<b>14</b>	<b>Computing the epsilon transitions for the composition of GTT's</b>	<b>136</b>
<b>15</b>	<b>Computing the epsilon transitions for the transitive closure of GTT's</b>	<b>137</b>

<b>16 Computing the epsilon transitions for the transitive closure of pair automata</b>	<b>139</b>
<b>17 Computing the Q infinity set for the infinity predicate automaton</b>	<b>140</b>

## 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* ( $\langle[- \leftarrow -]\rangle$  [1000, 0, 0] 1000) **where**  
  *replace-term-at*  $s \ [] \ t = t$   
  | *replace-term-at* (*Var*  $x$ )  $ps \ t = (\text{Var } x)$   
  | *replace-term-at* (*Fun*  $f \ ts$ ) ( $i \ \# \ ps$ )  $t =$   
    (*if*  $i < \text{length } ts$  *then* *Fun*  $f \ (ts[i:=(\text{replace-term-at } (ts \ ! \ i) \ ps \ t)])$  *else* *Fun*  $f \ ts$ )

**fun** *fun-at* :: ( $'f, 'v$ ) *term*  $\Rightarrow$  *pos*  $\Rightarrow$  ( $'f + 'v$ ) *option* **where**  
  *fun-at* (*Var*  $x$ )  $[] = \text{Some } (\text{Inr } x)$   
  | *fun-at* (*Fun*  $f \ ts$ )  $[] = \text{Some } (\text{Inl } f)$   
  | *fun-at* (*Fun*  $f \ ts$ ) ( $i \ \# \ p$ ) = (*if*  $i < \text{length } ts$  *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*  $f \equiv \text{map-prod } f \ f$

**definition** *adapt-vars* :: ( $'f, 'q$ ) *term*  $\Rightarrow$  ( $'f, 'v$ ) *term* **where**  
  *adapt-vars*  $\equiv \text{map-vars-term } (\lambda-. \text{undefined})$

**definition** *adapt-vars-ctxt* :: ( $'f, 'q$ ) *ctxt*  $\Rightarrow$  ( $'f, 'v$ ) *ctxt* **where**  
  *adapt-vars-ctxt* = *map-vars-ctxt*  $(\lambda-. \text{undefined})$

## 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**  $s \ |- \ p = t \ |- \ p \wedge \text{ctxt-of-pos-term } p \ s = \text{ctxt-of-pos-term } p \ t$  *<proof>*

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

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

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

*<proof>*

**lemma** *replace-term-at-replace-at-conv*:

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

*<proof>*

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

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

*<proof>*

**lemma** *replace-term-at-above* [simp]:  
 $p \leq_p q \implies s[q \leftarrow t][p \leftarrow u] = s[p \leftarrow u]$   
 ⟨proof⟩

**lemma** *replace-term-at-below* [simp]:  
 $p <_p q \implies s[p \leftarrow t][q \leftarrow u] = s[p \leftarrow t[q \leftarrow_p p \leftarrow u]]$   
 ⟨proof⟩

**lemma** *replace-at-hole-pos* [simp]:  $C \langle s \rangle [\text{hole-pos } C \leftarrow t] = C \langle t \rangle$   
 ⟨proof⟩

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

**lemma** *adapt-vars2*:  
 $\text{adapt-vars } (\text{adapt-vars } t) = \text{adapt-vars } t$   
 ⟨proof⟩

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

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

**lemma** *ground-adapt-vars* [simp]:  $\text{ground } (\text{adapt-vars } t) = \text{ground } t$   
 ⟨proof⟩

**lemma** *funas-term-adapt-vars*[simp]:  $\text{funas-term } (\text{adapt-vars } t) = \text{funas-term } t$   
 ⟨proof⟩

**lemma** *adapt-vars-adapt-vars*[simp]: **fixes**  $t :: ('f, 'v)\text{term}$   
**assumes**  $g: \text{ground } t$   
**shows**  $\text{adapt-vars } (\text{adapt-vars } t :: ('f, 'w)\text{term}) = t$   
 ⟨proof⟩

**lemma** *adapt-vars-inj*:  
**assumes**  $\text{adapt-vars } x = \text{adapt-vars } y \text{ ground } x \text{ ground } y$   
**shows**  $x = y$   
 ⟨proof⟩

**lemma** *adapt-vars-ctxt-simps*[simp, code]:  
 $\text{adapt-vars-ctxt } \text{Hole} = \text{Hole}$   
 $\text{adapt-vars-ctxt } (\text{More } f \text{ bef } C \text{ aft}) = \text{More } f \text{ (map adapt-vars bef) (adapt-vars-ctxt } C) \text{ (map adapt-vars aft)}$   
 ⟨proof⟩

**lemma** *adapt-vars-ctxt*[simp]:  $\text{adapt-vars } (C \langle t \rangle) = (\text{adapt-vars-ctxt } C) \langle \text{adapt-vars } t \rangle$   
 ⟨proof⟩

**lemma** *adapt-vars-subst*[simp]:  $\text{adapt-vars } (l \cdot \sigma) = l \cdot (\lambda x. \text{adapt-vars } (\sigma x))$   
 ⟨proof⟩

**lemma** *adapt-vars-gr-map-vars* [simp]:  
 $\text{ground } t \implies \text{map-vars-term } f t = \text{adapt-vars } t$   
 ⟨proof⟩

**lemma** *adapt-vars-gr-ctxt-of-map-vars* [simp]:  
 $\text{ground-ctxt } C \implies \text{map-vars-ctxt } f C = \text{adapt-vars-ctxt } C$   
 ⟨proof⟩

### 2.4.1 Equality on ground terms/contexts by positions and symbols

**lemma** *fun-at-def'*:  
 $\text{fun-at } t p = (\text{if } p \in \text{poss } t \text{ then}$   
    $(\text{case } t \text{ |- } p \text{ of Var } x \Rightarrow \text{Some } (\text{Inr } x) \mid \text{Fun } f \text{ ts} \Rightarrow \text{Some } (\text{Inl } f)) \text{ else None})$   
 ⟨proof⟩

**lemma** *fun-at-None-nposs-iff*:  
 $\text{fun-at } t p = \text{None} \iff p \notin \text{poss } t$   
 ⟨proof⟩

**lemma** *eq-term-by-poss-fun-at*:  
**assumes**  $\text{poss } s = \text{poss } t \wedge p. p \in \text{poss } s \implies \text{fun-at } s p = \text{fun-at } t p$   
**shows**  $s = t$   
 ⟨proof⟩

**lemma** *eq-ctxt-at-pos-by-poss*:  
**assumes**  $p \in \text{poss } s \wedge p \in \text{poss } t$   
**and**  $\bigwedge q. \neg (p \leq_p q) \implies q \in \text{poss } s \iff q \in \text{poss } t$   
**and**  $(\bigwedge q. q \in \text{poss } s \implies \neg (p \leq_p q) \implies \text{fun-at } s q = \text{fun-at } t q)$   
**shows**  $\text{ctxt-of-pos-term } p s = \text{ctxt-of-pos-term } p t$  ⟨proof⟩

## 2.5 Misc

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

**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⟩

**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:**  
 $\neg P (last xs) \implies filter-rev-nth P xs i = filter-rev-nth P (butlast xs) i$   
 $\langle proof \rangle$

**lemma filter-rev-nth-idx:**  
**assumes**  $i < length xs$   $P (xs ! i)$   $ys = filter P xs$   
**shows**  $xs ! i = ys ! (filter-rev-nth P xs i) \wedge filter-rev-nth P xs i < length ys$   
 $\langle proof \rangle$

**primrec add-elem-list-lists :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  '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:**  
 $ys \in set (add-elem-list-lists x xs) \implies length ys = Suc (length xs)$   
 $\langle proof \rangle$

**lemma add-elem-list-listsE:**  
**assumes**  $ys \in set (add-elem-list-lists x xs)$   
**shows**  $\exists n \leq length xs. ys = take n xs @ x \# drop n xs$   $\langle proof \rangle$

**lemma add-elem-list-listsI:**  
**assumes**  $n \leq length xs$   $ys = take n xs @ x \# drop n xs$   
**shows**  $ys \in set (add-elem-list-lists x xs)$   $\langle proof \rangle$

**lemma add-elem-list-lists-def':**  
 $set (add-elem-list-lists x xs) = \{ys \mid ys n. n \leq length xs \wedge ys = take n xs @ x \# drop n xs\}$   
 $\langle proof \rangle$

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

**lemma** *list-of-permutation-element-n-conv*:  
**assumes**  $n \neq 0$   
**shows** set (*list-of-permutation-element-n* x n L) =  
{xs | xs 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⟩

**lemma** *list-of-permutation-element-n-iff*:  
set (*list-of-permutation-element-n* x n L) =  
(if  $n = 0$  then {} else {xs | xs i.  $i < \text{length } xs \wedge (\forall j < \text{length } xs. j \neq i \longrightarrow xs ! j \in \text{set } L) \wedge \text{length } xs = n \wedge xs ! i = x$ } )  
⟨proof⟩

**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 ⊆ insert x (set L) ∧ length xs = n ∧ x ∈ set xs}  
⟨proof⟩

Misc

**lemma** *in-set-idx*:  
 $x \in \text{set } xs \implies \exists i < \text{length } xs. xs ! i = x$   
⟨proof⟩

**lemma** *set-list-subset-eq-nth-conv*:  
set xs ⊆ A ⟷ (∀ i < length xs. xs ! i ∈ A)  
⟨proof⟩

**lemma** *map-eq-nth-conv*:  
map f xs = map g ys ⟷ length xs = length ys ∧ (∀ i < length ys. f (xs ! i) = g (ys ! i))  
⟨proof⟩

**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 - Suc (length xs)))  
⟨proof⟩

**lemma** *map-prod-times*:  
f ' A × g ' B = map-prod f g ' (A × B)  
⟨proof⟩

**lemma** *trancl-full-on*:  $(X \times X)^+ = X \times X$

*<proof>*

**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**  $(f x, f y) \in s^+$  *<proof>*

**lemma** *trancl-map-prod-mono*:

$map\text{-}both\ f\ 'R^+ \subseteq (map\text{-}both\ f\ 'R)^+$   
*<proof>*

**lemma** *trancl-map-both-Restr*:

**assumes** *inj-on f X*  
**shows**  $(map\text{-}both\ f\ 'Restr\ R\ X)^+ = map\text{-}both\ f\ '(Restr\ R\ X)^+$   
*<proof>*

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

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

**lemma** *kleene-induct*:

$A \subseteq X \implies B\ O\ X \subseteq X \implies X\ O\ C \subseteq X \implies B^*\ O\ A\ O\ C^* \subseteq X$   
*<proof>*

**lemma** *kleene-trancl-induct*:

$A \subseteq X \implies B\ O\ X \subseteq X \implies X\ O\ C \subseteq X \implies B^+\ O\ A\ O\ C^+ \subseteq X$   
*<proof>*

**lemma** *rtrancl-Un2-separatorE*:

$B\ O\ A = \{\} \implies (A \cup B)^* = A^* \cup A^* O B^*$   
*<proof>*

**lemma** *trancl-Un2-separatorE*:

**assumes**  $B\ O\ A = \{\}$   
**shows**  $(A \cup B)^+ = A^+ \cup A^+ O B^+ \cup B^+$  (**is** ?Ls = ?Rs)  
*<proof>*

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

**lemma** *is-InrE*:

**assumes** *is-Inr q*  
**obtains** *p* **where**  $q = Inr\ p$   
*<proof>*

**lemma** *is-InlE*:

**assumes** *is-Inl q*  
**obtains** *p* **where**  $q = Inl\ p$   
*<proof>*

**lemma** *not-is-Inr-is-Inl* [simp]:

$\neg \text{is-Inl } t \longleftrightarrow \text{is-Inr } t$   
 $\neg \text{is-Inr } t \longleftrightarrow \text{is-Inl } t$   
<proof>

**lemma** [simp]: *remove-sum*  $\circ$  *Inl* = *id* <proof>

**abbreviation** *CInl* :: 'q  $\Rightarrow$  'q + 'q **where** *CInl*  $\equiv$  *Inl*

**abbreviation** *CInr* :: 'q  $\Rightarrow$  'q + 'q **where** *CInr*  $\equiv$  *Inr*

**lemma** *inj-CInl*: *inj CInl inj CInr* <proof>

**lemma** *map-prod-simp'*: *map-prod* *f g G* = (*f* (*fst G*), *g* (*snd G*))  
<proof>

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

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

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

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

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

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

*<proof>*

**lemma** *groot-sym-gterm-of-term*:

$ground\ t \implies groot\ sym\ (gterm\ of\ term\ t) = fst\ (the\ (root\ t))$

*<proof>*

**lemma** *length-args-length-gargs* [*simp*]:

$length\ (args\ (term\ of\ gterm\ t)) = length\ (gargs\ t)$

*<proof>*

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

$ground\ (term\ of\ gterm\ s)$

*<proof>*

**lemma** *ground-term-of-gterm'* [*simp*]:

$term\ of\ gterm\ s = Fun\ f\ ss \implies ground\ (Fun\ f\ ss)$

*<proof>*

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

$gterm\ of\ term\ (term\ of\ gterm\ t) = t$

*<proof>*

**lemma** *inj-term-of-gterm*:

*inj-on term-of-gterm X*

*<proof>*

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

$ground\ t \implies term\ of\ gterm\ (gterm\ of\ term\ t) = t$

*<proof>*

**lemma** *ground-term-to-gtermD*:

$ground\ t \implies \exists t'. t = term\ of\ gterm\ t'$

*<proof>*

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

$map\ term\ f\ g\ (term\ of\ gterm\ t) = term\ of\ gterm\ (map\ gterm\ f\ t)$

*<proof>*

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

$ground\ t \implies gterm\ of\ term\ (map\ term\ f\ g\ t) = map\ gterm\ f\ (gterm\ of\ term\ t)$

*<proof>*

**lemma** *gterm-set-gterm-funs-terms*:

$set\ gterm\ t = funs\ term\ (term\ of\ gterm\ t)$

*<proof>*

**lemma** *term-set-gterm-funs-terms*:

**assumes**  $ground\ t$

**shows**  $set\ gterm\ (gterm\ of\ term\ t) = funs\ term\ t$

$\langle \text{proof} \rangle$

**lemma** *vars-term-of-gterm* [simp]:  
vars-term (term-of-gterm t) = {}  
 $\langle \text{proof} \rangle$

**lemma** *vars-term-of-gterm-subseteq* [simp]:  
vars-term (term-of-gterm t)  $\subseteq$  Q  $\longleftrightarrow$  True  
 $\langle \text{proof} \rangle$

**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  
 $\langle \text{proof} \rangle$

**lemma** *gposs-map-gterm* [simp]:  
gposs (map-gterm f s) = gposs s  
 $\langle \text{proof} \rangle$

**lemma** *poss-gposs-conv*:  
poss (term-of-gterm t) = gposs t  
 $\langle \text{proof} \rangle$

**lemma** *poss-gposs-mem-conv*:  
p  $\in$  poss (term-of-gterm t)  $\longleftrightarrow$  p  $\in$  gposs t  
 $\langle \text{proof} \rangle$

**lemma** *gposs-to-poss*:  
p  $\in$  gposs t  $\Longrightarrow$  p  $\in$  poss (term-of-gterm t)  
 $\langle \text{proof} \rangle$

**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-sum ( $\lambda$  x. x) ( $\lambda$  -.undefined)

**lemma** *gfun-at-gterm-of-term* [simp]:  
ground s  $\Longrightarrow$  map-option exInl (fun-at s p) = gfun-at (gterm-of-term s) p  
 $\langle \text{proof} \rangle$

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

**lemma** *gfun-at-None-ngposs-iff*:  $gfun-at\ s\ p = None \longleftrightarrow p \notin gposs\ s$   
 ⟨proof⟩

**lemma** *gfun-at-map-gterm* [simp]:  
 $gfun-at\ (map-gterm\ f\ t)\ p = map-option\ f\ (gfun-at\ t\ p)$   
 ⟨proof⟩

**lemma** *set-gterm-gposs-conv*:  
 $set-gterm\ t = \{the\ (gfun-at\ t\ p) \mid p.\ p \in gposs\ t\}$   
 ⟨proof⟩

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⟩

**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**  
 $gsubt-at\ s\ [] = s \mid$   
 $gsubt-at\ (GFun\ f\ ss)\ (i\ \# \ p) = gsubt-at\ (ss\ !\ i)\ p$

**lemma** *gsubt-at-to-subt-at*:  
**assumes**  $p \in gposs\ s$   
**shows**  $gterm-of-term\ (term-of-gterm\ s\ |- \ p) = gsubt-at\ s\ p$   
 ⟨proof⟩

**lemma** *term-of-gterm-gsubt*:  
**assumes**  $p \in gposs\ s$   
**shows**  $(term-of-gterm\ s)\ |- \ p = term-of-gterm\ (gsubt-at\ s\ p)$   
 ⟨proof⟩

**lemma** *gsubt-at-gposs* [simp]:  
**assumes**  $p \in gposs\ s$   
**shows**  $gposs\ (gsubt-at\ s\ p) = \{x \mid x.\ p\ @\ x \in gposs\ s\}$   
 ⟨proof⟩

**lemma** *gfun-at-gsub-at* [simp]:  
**assumes**  $p \in gposs\ s$  **and**  $p\ @\ q \in gposs\ s$   
**shows**  $gfun-at\ (gsubt-at\ s\ p)\ q = gfun-at\ s\ (p\ @\ q)$   
 ⟨proof⟩

**lemma** *gposs-gsubst-at-subst-at-eq* [simp]:  
**assumes**  $p \in gposs\ s$

**shows**  $gposs (gsubt-at s p) = poss (term-of-gterm s \mid - p)$   $\langle proof \rangle$

**lemma** *gpos-append-gposs*:

**assumes**  $p \in gposs t$  **and**  $q \in gposs (gsubt-at t p)$

**shows**  $p @ q \in gposs t$

$\langle proof \rangle$

Replace terms at position

**fun** *replace-gterm-at* ( $\langle \cdot \mid - \leftarrow \cdot \rangle_G$  [1000, 0, 0] 1000) **where**

*replace-gterm-at*  $s \ [] \ t = t$

$\mid$  *replace-gterm-at* ( $GFun f ts$ ) ( $i \# ps$ )  $t =$

(if  $i < length\ ts$  then  $GFun f (ts[i:= (replace-gterm-at (ts ! i) ps t)])$  else  $GFun f ts$ )

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

$p \notin gposs s \implies s[p \leftarrow t]_G = s$

$\langle proof \rangle$

**lemma** *parallel-replace-gterm-commute* [ac-simps]:

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

$\langle proof \rangle$

**lemma** *replace-gterm-at-above* [simp]:

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

$\langle proof \rangle$

**lemma** *replace-gterm-at-below* [simp]:

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

$\langle proof \rangle$

**lemma** *groot-sym-replace-gterm* [simp]:

$p \neq [] \implies groot-sym\ s[p \leftarrow t]_G = groot-sym\ s$

$\langle proof \rangle$

**lemma** *replace-gterm-gsubt-at-id* [simp]:  $s[p \leftarrow gsubt-at s p]_G = s$

$\langle proof \rangle$

**lemma** *replace-gterm-conv*:

$p \in gposs s \implies (term-of-gterm s)[p \leftarrow (term-of-gterm t)] = term-of-gterm (s[p \leftarrow t]_G)$

$\langle proof \rangle$

## 2.6.2 Tree domains

**type-synonym** *gdomain* = *unit gterm*

**abbreviation** *gdomain* **where**

*gdomain*  $\equiv map-gterm (\lambda \cdot. ())$

**lemma** *gdomain-id*:

$gdomain\ t = t$   
 $\langle proof \rangle$

**lemma** *gdomain-gsubt* [*simp*]:  
assumes  $p \in gposs\ t$   
shows  $gdomain\ (gsubt\text{-}at\ t\ p) = gsubt\text{-}at\ (gdomain\ t)\ p$   
 $\langle proof \rangle$

Union of tree domains

**fun** *gunion* ::  $gdomain \Rightarrow gdomain \Rightarrow gdomain$  **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*]:  
 $gposs\ (gunion\ s\ t) = gposs\ s \cup gposs\ t$   
 $\langle proof \rangle$

**lemma** *gunion-unit* [*simp*]:  
 $gunion\ s\ (GFun\ ()\ []) = s\ gunion\ (GFun\ ()\ [])\ s = s$   
 $\langle proof \rangle$

**lemma** *gunion-gsubt-at-nt-poss1*:  
assumes  $p \in gposs\ s$  **and**  $p \notin gposs\ t$   
shows  $gsubt\text{-}at\ (gunion\ s\ t)\ p = gsubt\text{-}at\ s\ p$   
 $\langle proof \rangle$

**lemma** *gunion-gsubt-at-nt-poss2*:  
assumes  $p \in gposs\ t$  **and**  $p \notin gposs\ s$   
shows  $gsubt\text{-}at\ (gunion\ s\ t)\ p = gsubt\text{-}at\ t\ p$   
 $\langle proof \rangle$

**lemma** *gunion-gsubt-at-poss*:  
assumes  $p \in gposs\ s$  **and**  $p \in gposs\ t$   
shows  $gunion\ (gsubt\text{-}at\ s\ p)\ (gsubt\text{-}at\ t\ p) = gsubt\text{-}at\ (gunion\ s\ t)\ p$   
 $\langle proof \rangle$

**lemma** *gfun-at-domain*:  
shows  $gfun\text{-}at\ t\ p = (if\ p \in gposs\ t\ then\ Some\ ()\ else\ None)$   
 $\langle proof \rangle$

**lemma** *gunion-assoc* [*ac-simps*]:  
 $gunion\ s\ (gunion\ t\ u) = gunion\ (gunion\ s\ t)\ u$   
 $\langle proof \rangle$

**lemma** *gunion-commute* [*ac-simps*]:  
 $gunion\ s\ t = gunion\ t\ s$   
 $\langle proof \rangle$

**lemma** *gunion-idemp* [simp]:

*gunion s s = s*  
*<proof>*

**definition** *gunions* :: *gdomain list*  $\Rightarrow$  *gdomain* **where**

*gunions ts = foldr gunion ts (GFun () [])*

**lemma** *gunions-append*:

*gunions (ss @ ts) = gunion (gunions ss) (gunions ts)*  
*<proof>*

**lemma** *gposs-gunions* [simp]:

*gposs (gunions ts) = {[]}  $\cup$   $\bigcup$  {gposs t | t. t  $\in$  set ts}*  
*<proof>*

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  $\circ$  (#) i) (ts ! i)) [0..*length* ts])*

**end**

**lemma** *map-gterm-glabel*:

*map-gterm f (glabel h t) = glabel (f  $\circ$  h) t*  
*<proof>*

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

*gfun-at (glabel f t) p = (if p  $\in$  gposs t then Some (f p) else None)*  
*<proof>*

**lemma** *gposs-glabel* [simp]:

*gposs (glabel f t) = gposs t*  
*<proof>*

**lemma** *glabel-map-gterm-conv*:

*glabel (f  $\circ$  gfun-at t) (gdomain t) = map-gterm (f  $\circ$  Some) t*  
*<proof>*

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

*p  $\notin$  gposs t  $\implies$  gfun-at t p = None*  
*<proof>*

**lemma** *gfun-at-poss*:

*p  $\in$  gposs t  $\implies$   $\exists$ f. gfun-at t p = Some f*  
*<proof>*

**lemma** *gfun-at-possE*:

**assumes**  $p \in gposs\ t$

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

$\langle proof \rangle$

**lemma** *gfun-at-poss-gpossD*:

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

$\langle proof \rangle$

function symbols of a ground term

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

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

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

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

$\langle proof \rangle$

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

$funas-term\ (term-of-gterm\ t) = funas-gterm\ t$

$\langle proof \rangle$

**lemma** *funas-gterm-map-gterm*:

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

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

$\langle proof \rangle$

**lemma** *gterm-of-term-inj*:

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

**shows** *inj-on*  $gterm-of-term\ S$

$\langle proof \rangle$

**lemma** *funas-gterm-gsubt-at-subseteq*:

**assumes**  $p \in gposs\ s$

**shows**  $funas-gterm\ (gsubt-at\ s\ p) \subseteq funas-gterm\ s$   $\langle proof \rangle$

**lemma** *finite-funas-gterm*:  $finite\ (funas-gterm\ t)$

$\langle proof \rangle$

ground term set

**abbreviation** *gterms* **where**

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

**lemma** *gterms-mono*:

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

$\langle proof \rangle$

**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]:*  $(f, n) \in \mathcal{F} \implies \text{length } ss = n \implies (\bigwedge i. i < \text{length } ss \implies ss ! i \in \mathcal{T}_G \mathcal{F}) \implies G\text{Fun } f \text{ } ss \in \mathcal{T}_G \mathcal{F}$

**lemma**  *$\mathcal{T}_G$ -sound:*

$s \in \mathcal{T}_G \mathcal{F} \implies \text{funas-gterm } s \subseteq \mathcal{F}$   
 ⟨proof⟩

**lemma**  *$\mathcal{T}_G$ -complete:*

$\text{funas-gterm } s \subseteq \mathcal{F} \implies s \in \mathcal{T}_G \mathcal{F}$   
 ⟨proof⟩

**lemma**  *$\mathcal{T}_G$ -funas-gterm-conv:*

$s \in \mathcal{T}_G \mathcal{F} \longleftrightarrow \text{funas-gterm } s \subseteq \mathcal{F}$   
 ⟨proof⟩

**lemma**  *$\mathcal{T}_G$ -equivalent-def:*

$\mathcal{T}_G \mathcal{F} = \text{gterms } \mathcal{F}$   
 ⟨proof⟩

**lemma**  *$\mathcal{T}_G$ -intersection [simp]:*

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

**lemma**  *$\mathcal{T}_G$ -mono:*

$\mathcal{G} \subseteq \mathcal{F} \implies \mathcal{T}_G \mathcal{G} \subseteq \mathcal{T}_G \mathcal{F}$   
 ⟨proof⟩

**lemma**  *$\mathcal{T}_G$ -UNIV [simp]:*  $s \in \mathcal{T}_G \text{ UNIV}$

⟨proof⟩

**definition** *funas-grel where*

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

**end**

**theory** *FSet-Utills*

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

**begin**

**context**

**includes** *fset.lifting*

**begin**

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

⟨proof⟩

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

$\langle proof \rangle$

**lift-definition** *is-empty* :: 'a fset  $\Rightarrow$  bool **is** *Set.is-empty*  $\langle proof \rangle$

**lift-definition** *fremove* :: 'a  $\Rightarrow$  'a fset  $\Rightarrow$  'a fset **is** *Set.remove*  
 $\langle proof \rangle$

**lift-definition** *finj-on* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a fset  $\Rightarrow$  bool **is** *inj-on*  $\langle proof \rangle$

**lift-definition** *the-finv-into* :: 'a fset  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  'a **is** *the-inv-into*  
 $\langle proof \rangle$

**lemma** *fCollect-memberI* [*intro!*]:

*finite* (Collect *P*)  $\Longrightarrow$  *P* *x*  $\Longrightarrow$  *x*  $\in$  | *fCollect* *P*  
 $\langle proof \rangle$

**lemma** *fCollect-member* [*iff*]:

*x*  $\in$  | *fCollect* *P*  $\longleftrightarrow$  *finite* (Collect *P*)  $\wedge$  *P* *x*  
 $\langle proof \rangle$

**lemma** *fCollect-cong*: ( $\wedge x. P\ x = Q\ x$ )  $\Longrightarrow$  *fCollect* *P* = *fCollect* *Q*  
 $\langle proof \rangle$

**end**

**syntax**

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

**syntax-consts**

*-fColl*  $\Rightarrow$  *fCollect*

**translations**

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

**syntax** (*ASCII*)

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

**syntax**

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

**syntax-consts**

*-fCollect*  $\Rightarrow$  *fCollect*

**translations**

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

**syntax**

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

**syntax-consts**

*-fSetcompr*  $\Rightarrow$  *fCollect*

$\langle ML \rangle$

**syntax**

*-fSigma* :: *pttrn*  $\Rightarrow$  'a fset  $\Rightarrow$  'b fset  $\Rightarrow$  ('a  $\times$  'b) set ( $\langle (3\SIGMA\ -|:-./-\) \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**\langle proof \rangle***lemma** *bi-unique-cr-fset* [*transfer-rule*]:*bi-unique cr-fset**\langle proof \rangle***lemma** *right-total-pcr-fset-eq* [*transfer-rule*]:*right-total (pcr-fset (=))**\langle proof \rangle***lemma** *bi-unique-pcr-fset* [*transfer-rule*]:*bi-unique (pcr-fset (=))**\langle proof \rangle***lemma** *set-fset-of-list-transfer* [*transfer-rule*]:*rel-fun (list-all2 A) (pcr-fset A) set fset-of-list**\langle proof \rangle***lemma** *fCollectD*:  $a \in | \{ | x . P\ x \} \implies P\ a$ *\langle proof \rangle***lemma** *fCollectI*:  $P\ a \implies finite\ (Collect\ P) \implies a \in | \{ | x. P\ x \}$ *\langle proof \rangle***lemma** *fCollect-fempty-eq* [*simp*]:  $fCollect\ P = \{ | \} \iff (\forall x. \neg P\ x) \vee infinite\ (Collect\ P)$ *\langle proof \rangle***lemma** *fempty-fCollect-eq* [*simp*]:  $\{ | \} = fCollect\ P \iff (\forall x. \neg P\ x) \vee infinite\ (Collect\ P)$ *\langle proof \rangle*

**lemma** *fset-image-conv*:

$$\{f\ x \mid x.\ x \in T\} = \text{fset } (f \mid \uparrow T)$$

*<proof>*

**lemma** *fimage-def*:

$$f \mid \uparrow A = \{ \mid y.\ \exists x \in A.\ y = f\ x \}$$

*<proof>*

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

*<proof>*

**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  $A \mid \times \mid B \equiv \text{fSigma } A\ (\lambda\cdot.\ B)$

**lemma** *fSigma-repeq*:

$$\text{fset } (A \mid \times \mid B) = \text{fset } A \times \text{fset } B$$

*<proof>*

**lemmas** *fSigmaI* [*intro!*] = *SigmaI*[*Transfer.transferred*]

**lemmas** *fSigmaE* [*elim!*] = *SigmaE*[*Transfer.transferred*]

**lemmas** *fSigmaD1* = *SigmaD1*[*Transfer.transferred*]

**lemmas** *fSigmaD2* = *SigmaD2*[*Transfer.transferred*]

**lemmas** *fSigmaE2* = *SigmaE2*[*Transfer.transferred*]

**lemmas** *fSigma-cong* = *Sigma-cong*[*Transfer.transferred*]

**lemmas** *fSigma-mono* = *Sigma-mono*[*Transfer.transferred*]

**lemmas** *fSigma-empty1* [*simp*] = *Sigma-empty1*[*Transfer.transferred*]

**lemmas** *fSigma-empty2* [*simp*] = *Sigma-empty2*[*Transfer.transferred*]

**lemmas** *fmem-Sigma-iff* [*iff*] = *mem-Sigma-iff*[*Transfer.transferred*]

**lemmas** *fmem-Times-iff* = *mem-Times-iff*[*Transfer.transferred*]

**lemmas** *fSigma-empty-iff* = *Sigma-empty-iff*[*Transfer.transferred*]

**lemmas** *fTimes-subset-cancel2* = *Times-subset-cancel2*[*Transfer.transferred*]

**lemmas** *fTimes-eq-cancel2* = *Times-eq-cancel2*[*Transfer.transferred*]

**lemmas** *fUN-Times-distrib* = *UN-Times-distrib*[*Transfer.transferred*]

**lemmas** *fsplit-paired-Ball-Sigma* [*simp*, *no-atp*] = *split-paired-Ball-Sigma*[*Transfer.transferred*]

**lemmas** *fsplit-paired-Bex-Sigma* [*simp*, *no-atp*] = *split-paired-Bex-Sigma*[*Transfer.transferred*]

**lemmas** *fSigma-Un-distrib1* = *Sigma-Un-distrib1*[*Transfer.transferred*]

**lemmas** *fSigma-Un-distrib2* = *Sigma-Un-distrib2*[*Transfer.transferred*]

**lemmas** *fSigma-Int-distrib1* = *Sigma-Int-distrib1*[*Transfer.transferred*]

**lemmas** *fSigma-Int-distrib2* = *Sigma-Int-distrib2*[*Transfer.transferred*]

**lemmas** *fSigma-Diff-distrib1* = *Sigma-Diff-distrib1*[*Transfer.transferred*]

**lemmas** *fSigma-Diff-distrib2* = *Sigma-Diff-distrib2*[*Transfer.transferred*]

**lemmas** *fSigma-Union* = *Sigma-Union*[*Transfer.transferred*]

**lemmas** *fTimes-Un-distrib1* = *Times-Un-distrib1*[*Transfer.transferred*]

**lemmas** *fTimes-Int-distrib1* = *Times-Int-distrib1*[*Transfer.transferred*]

**lemmas** *fTimes-Diff-distrib1* = *Times-Diff-distrib1*[*Transfer.transferred*]

**lemmas** *fTimes-empty* [simp] = *Times-empty*[*Transfer.transferred*]  
**lemmas** *ftimes-subset-iff* = *times-subset-iff*[*Transfer.transferred*]  
**lemmas** *ftimes-eq-iff* = *times-eq-iff*[*Transfer.transferred*]  
**lemmas** *ffst-image-times* [simp] = *fst-image-times*[*Transfer.transferred*]  
**lemmas** *fsnd-image-times* [simp] = *snd-image-times*[*Transfer.transferred*]  
**lemmas** *fsnd-image-Sigma* = *snd-image-Sigma*[*Transfer.transferred*]  
**lemmas** *finsert-Times-insert* = *insert-Times-insert*[*Transfer.transferred*]  
**lemmas** *fTimes-Int-Times* = *Times-Int-Times*[*Transfer.transferred*]  
**lemmas** *fiimage-paired-Times* = *image-paired-Times*[*Transfer.transferred*]  
**lemmas** *fproduct-swap* = *product-swap*[*Transfer.transferred*]  
**lemmas** *fswap-product* = *swap-product*[*Transfer.transferred*]  
**lemmas** *fsubset-fst-snd* = *subset-fst-snd*[*Transfer.transferred*]  
**lemmas** *map-prod-ftimes* = *map-prod-times*[*Transfer.transferred*]

**lemma** *fCollect-case-prod* [simp]:  
 $\{|(a, b). P a \wedge Q b|\} = fCollect P \times | fCollect Q$   
*<proof>*  
**lemma** *fCollect-case-prodD*:  
 $x \in | \{|(x, y). A x y|\} \implies A (fst x) (snd x)$   
*<proof>*

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

— Dealing with transitive closure

**lift-definition** *ftrancl* :: ('a × 'a) fset ⇒ ('a × 'a) fset (*<(-|+)>*) [1000] 999 **is**  
*trancl*  
*<proof>*

**lemmas** *fr-into-trancl* [intro, *Pure.intro*] = *r-into-trancl*[*Transfer.transferred*]  
**lemmas** *ftrancl-into-trancl* [*Pure.intro*] = *trancl-into-trancl*[*Transfer.transferred*]  
**lemmas** *ftrancl-induct*[*consumes 1, case-names Base Step*] = *trancl.induct*[*Transfer.transferred*]  
**lemmas** *ftrancl-mono* = *trancl-mono*[*Transfer.transferred*]  
**lemmas** *ftrancl-trans*[*trans*] = *trancl-trans*[*Transfer.transferred*]  
**lemmas** *ftrancl-empty* [simp] = *trancl-empty* [*Transfer.transferred*]  
**lemmas** *ftranclE*[*cases set: ftrancl*] = *tranclE*[*Transfer.transferred*]  
**lemmas** *converse-ftrancl-induct*[*consumes 1, case-names Base Step*] = *converse-trancl-induct*[*Transfer.transferred*]  
**lemmas** *converse-ftranclE* = *converse-tranclE*[*Transfer.transferred*]  
**lemma** *in-ftrancl-UnI*:  
 $x \in | R|^+ \vee x \in | S|^+ \implies x \in | (R \cup S)|^+$   
*<proof>*

**lemma** *ftranclD*:

$(x, y) \in R^+ \implies \exists z. (x, z) \in R \wedge (z = y \vee (z, y) \in R^+)$   
 ⟨proof⟩

**lemma** *ftranclD2*:

$(x, y) \in R^+ \implies \exists z. (x = z \vee (x, z) \in R^+) \wedge (z, y) \in R$   
 ⟨proof⟩

**lemma** *not-ftrancl-into*:

$(x, z) \notin r^+ \implies (y, z) \in r \implies (x, y) \notin r^+$   
 ⟨proof⟩

**lemmas** *ftrancl-map-both-fRestr* = *trancl-map-both-Restr*[*Transfer.transferred*]

**lemma** *ftrancl-map-both-fsubset*:

$f\text{inj-on } f \ X \implies R \subseteq X \times X \implies (\text{map-both } f \ \upharpoonright \ R)^+ = \text{map-both } f \ \upharpoonright \ R^+$   
 ⟨proof⟩

**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*:

$f\text{set-of-list } [i..<j] = f\text{Collect } (\lambda n. i \leq n \wedge n < j)$   
 ⟨proof⟩

**abbreviation** *fRestr* :: ('a × 'a) fset ⇒ 'a fset ⇒ ('a × 'a) fset **where**

$f\text{Restr } r \ A \equiv r \ \upharpoonright \ (A \times A)$

**lift-definition** *fId-on* :: 'a fset ⇒ ('a × 'a) fset **is** *Id-on*

⟨proof⟩

**lemmas** *fId-on-empty* [simp] = *Id-on-empty* [*Transfer.transferred*]

**lemmas** *fId-on-eqI* = *Id-on-eqI* [*Transfer.transferred*]

**lemmas** *fId-onI* [intro!] = *Id-onI* [*Transfer.transferred*]

**lemmas**  $fId\text{-}onE$  [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 × 'b) fset ⇒ ('b × 'a) fset (⟨(|<sup>-1</sup>)⟩ [1000] 999)  
**is**  $converse$  ⟨proof⟩

**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  
⟨proof⟩

**lemma**  $finj\text{-}CInl\text{-}CInr$ :  
 $finj\text{-}on$  CInl A  $finj\text{-}on$  CInr A  
⟨proof⟩

**lemma**  $finj\text{-}Some$ :  
 $finj\text{-}on$  Some A  
⟨proof⟩

**lift-definition**  $fImage$  :: ('a × 'b) fset ⇒ 'a fset ⇒ 'b fset (**infixr** ⟨|'⟩ 90) **is**  
 $Image$   
⟨proof⟩

**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**  $rel\text{-}set\text{-}cr\text{-}fset$ :  
 $rel\text{-}set\ cr\text{-}fset = (\lambda A B. A = fset \text{ ' } B)$   
 <proof>

**lemma**  $pcr\text{-}fset\text{-}cr\text{-}fset$ :  
 $pcr\text{-}fset\ cr\text{-}fset = (\lambda x y. x = fset (fset \text{ |' } y))$   
 <proof>

**lemma**  $sorted\text{-}list\text{-}of\text{-}fset\text{-}id$ :  
 $sorted\text{-}list\text{-}of\text{-}fset\ x = sorted\text{-}list\text{-}of\text{-}fset\ y \implies x = y$   
 <proof>

**lemmas**  $fBall\text{-}def = Ball\text{-}def$  [Transfer.transferred]  
**lemmas**  $fBex\text{-}def = Bex\text{-}def$  [Transfer.transferred]  
**lemmas**  $fCollectE = fCollectD$  [elim-format]  
**lemma**  $fCollect\text{-}conj\text{-}eq$ :  
 $finite (Collect P) \implies finite (Collect Q) \implies \{x. P\ x \wedge Q\ x\} = fCollect\ P \text{ | } \cap \text{ | } fCollect\ Q$   
 <proof>

**lemma** *finite-ntrancl*:

$finite\ R \implies finite\ (ntrancl\ n\ R)$   
*<proof>*

**lift-definition** *nfttrancl* ::  $nat \Rightarrow ('a \times 'a)\ fset \Rightarrow ('a \times 'a)\ fset$  **is** *ntrancl*  
*<proof>*

**lift-definition** *frelcomp* ::  $('a \times 'b)\ fset \Rightarrow ('b \times 'c)\ fset \Rightarrow ('a \times 'c)\ fset$  (**infixr**  $\langle |O| \rangle$  75) **is** *relcomp*  
*<proof>*

**lemmas** *frelcompE[elim!]* = *relcompE[Transfer.transferred]*

**lemmas** *frelcompI[intro]* = *relcompI[Transfer.transferred]*

**lemma** *fId-on-frelcomp-id*:

$fst\ |\cdot| R\ |\subseteq| S \implies fId-on\ S\ |O|\ R = R$   
*<proof>*

**lemma** *fId-on-frelcomp-id2*:

$snd\ |\cdot| R\ |\subseteq| S \implies R\ |O|\ fId-on\ S = R$   
*<proof>*

**lemmas** *fimage-fset* = *image-set[Transfer.transferred]*

**lemmas** *ftrancl-Un2-separatorE* = *trancl-Un2-separatorE[Transfer.transferred]*

**lemma** *finite-funs-term*:  $finite\ (funs-term\ t)$  *<proof>*

**lemma** *finite-funas-term*:  $finite\ (funas-term\ t)$  *<proof>*

**lemma** *finite-vars-ctxt*:  $finite\ (vars-ctxt\ C)$  *<proof>*

**lift-definition** *ffuns-term* ::  $('f, 'v)\ term \Rightarrow 'f\ fset$  **is** *funs-term* *<proof>*

**lift-definition** *fvars-term* ::  $('f, 'v)\ term \Rightarrow 'v\ fset$  **is** *vars-term* *<proof>*

**lift-definition** *fvars-ctxt* ::  $('f, 'v)\ ctxt \Rightarrow 'v\ fset$  **is** *vars-ctxt* *<proof>*

**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]*:  $ffuns-term\ (Var\ x) = \{|\}\}$   
*<proof>*

**lemma** *ffuns-term-Fun [simp]*:  $ffuns-term\ (Fun\ f\ ts) = |\cup| (ffuns-term\ |\cdot|\ fset-of-list\ ts)\ |\cup|\ \{|\cdot|\}$   
*<proof>*

**lemma** *fvars-term-Var [simp]*:  $fvars-term\ (Var\ x) = \{|\cdot|\}$   
*<proof>*

**lemma** *fvars-term-Fun [simp]*:  $fvars-term\ (Fun\ f\ ts) = |\cup| (fvars-term\ |\cdot|\ fset-of-list\ ts)\ |\cup|\ \{|\cdot|\}$

*ts*)  
⟨*proof*⟩

**lift-definition** *ffunas-term* :: ('f, 'v) term ⇒ ('f × nat) fset **is** funas-term  
⟨*proof*⟩

**lift-definition** *ffunas-gterm* :: 'f gterm ⇒ ('f × nat) fset **is** funas-gterm  
⟨*proof*⟩

**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)))

⟨*proof*⟩

**end**

**lemma** *finite-snd* [*intro*]:

*finite* S ⇒ *finite* {x. (y, x) ∈ S}

⟨*proof*⟩

**lemma** *finite-Collect-less-eq*:

$Q \leq P \implies \text{finite } (\text{Collect } P) \implies \text{finite } (\text{Collect } Q)$

⟨*proof*⟩

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

**lemma** *inj-FSet-Lex-Wrapper*: *inj* *Wrapp*

⟨*proof*⟩

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

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

**begin**

**definition** *less-eq-FSet-Lex-Wrapper* :: ('a :: *linorder*) *FSet-Lex-Wrapper* ⇒ 'a  
*FSet-Lex-Wrapper* ⇒ *bool*

**where** *less-eq-FSet-Lex-Wrapper* S T =

(*let* S' = *sorted-list-of-fset* (*ex* S) *in*

*let* T' = *sorted-list-of-fset* (*ex* T) *in*

S' ≤ T')

**definition** *less-FSet-Lex-Wrapper* :: 'a *FSet-Lex-Wrapper* ⇒ 'a *FSet-Lex-Wrapper*  
⇒ *bool*

```

where less-FSet-Lex-Wrapper  $S\ T =$ 
  (let  $S' = \text{sorted-list-of-fset } (ex\ S)$  in
   let  $T' = \text{sorted-list-of-fset } (ex\ T)$  in
    $S' < T'$ )

instance  $\langle proof \rangle$ 
end

end
theory Ground-Ctxt
  imports Ground-Terms
begin

2.6.3 Ground context

datatype (gfun-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  $f\ ss1\ C\ ss2$ )  $\langle s \rangle_G = \text{GFun } f\ (ss1\ @\ C\ \langle s \rangle_G\ \#\ ss2)$ 

fun hole-gpos where
  hole-gpos  $\square_G = []$  |
  hole-gpos (GMore  $f\ ss1\ C\ ss2$ ) = length  $ss1\ \#$  hole-gpos  $C$ 

lemma gctxt-eq [simp]: ( $C\ \langle s \rangle_G = C\ \langle t \rangle_G$ ) = ( $s = t$ )
   $\langle proof \rangle$ 

fun gctxt-compose :: 'f gctxt  $\Rightarrow$  'f gctxt  $\Rightarrow$  'f gctxt (infixl  $\langle \circ_{G_c} \rangle$  75) where
   $\square_G \circ_{G_c} D = D$  |
  (GMore  $f\ ss1\ C\ ss2$ )  $\circ_{G_c} D = \text{GMore } f\ ss1\ (C\ \circ_{G_c} D)\ ss2$ 

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

interpretation ctxt-monoid-mult: monoid-mult  $\square_G\ (\circ_{G_c})$ 
   $\langle proof \rangle$ 

instantiation gctxt :: (type) monoid-mult
begin
  definition [simp]:  $1 = \square_G$ 
  definition [simp]:  $(*) = (\circ_{G_c})$ 
  instance  $\langle proof \rangle$ 

```

**end**

**lemma** *ctxt-ctxt-compose* [simp]:  $(C \circ_{Gc} D)\langle t \rangle_G = C\langle D\langle t \rangle_G \rangle_G$   
*<proof>*

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

**fun** *ctxt-of-gctxt* **where**

*ctxt-of-gctxt*  $\square_G = \square$   
| *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*  $\square = \square_G$   
| *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*)  
*<proof>*

**lemma** *ground-ctxt-of-gctxt'* [simp]:

*ctxt-of-gctxt* *C* = *More* *f* *ss* *D* *ts*  $\implies$  *ground-ctxt* (*More* *f* *ss* *D* *ts*)  
*<proof>*

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

*gctxt-of-ctxt* (*ctxt-of-gctxt* *t*) = *t*  
*<proof>*

**lemma** *inj-ctxt-of-gctxt*: *inj-on* *ctxt-of-gctxt* *X*

*<proof>*

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

*ground-ctxt* *C*  $\implies$  *ctxt-of-gctxt* (*gctxt-of-ctxt* *C*) = *C*  
*<proof>*

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

*map-ctxt* *f* *g* (*ctxt-of-gctxt* *C*) = *ctxt-of-gctxt* (*map-gctxt* *f* *C*)  
*<proof>*

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

*ground-ctxt* *C*  $\implies$  *gctxt-of-ctxt* (*map-ctxt* *f* *g* *C*) = *map-gctxt* *f* (*gctxt-of-ctxt* *C*)  
*<proof>*

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

*C*  $\neq \square_G \implies$  *map-gctxt* *f* *C*  $\neq \square_G$   
*<proof>*

**lemma** *gctxt-set-funs-ctxt*:

$gfuns-ctxt\ C = funs-ctxt\ (ctxt-of-gctxt\ C)$   
 $\langle proof \rangle$

**lemma** *ctxt-set-funs-gctxt*:  
**assumes** *ground-ctxt*  $C$   
**shows**  $gfuns-ctxt\ (gctxt-of-ctxt\ C) = funs-ctxt\ C$   
 $\langle proof \rangle$

**lemma** *vars-ctxt-of-gctxt* [*simp*]:  
 $vars-ctxt\ (ctxt-of-gctxt\ C) = \{\}$   
 $\langle proof \rangle$

**lemma** *vars-ctxt-of-gctxt-subseteq* [*simp*]:  
 $vars-ctxt\ (ctxt-of-gctxt\ C) \subseteq Q \iff True$   
 $\langle proof \rangle$

**lemma** *term-of-gterm-ctxt-apply-ground* [*simp*]:  
 $term-of-gterm\ s = C\langle l \rangle \implies ground-ctxt\ C$   
 $term-of-gterm\ s = C\langle l \rangle \implies ground\ l$   
 $\langle proof \rangle$

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

**lemma** *gctxt-compose-HoleE*:  
 $C \circ_{Gc}\ D = \square_G \implies C = \square_G$   
 $C \circ_{Gc}\ D = \square_G \implies D = \square_G$   
 $\langle proof \rangle$

**lemma** *nempty-ground-ctxt-gctxt* [*simp*]:  
 $C \neq \square \implies ground-ctxt\ C \implies gctxt-of-ctxt\ C \neq \square_G$   
 $\langle proof \rangle$

**lemma** *ctxt-of-gctxt-apply* [*simp*]:  
 $gterm-of-term\ (ctxt-of-gctxt\ C)\langle term-of-gterm\ t \rangle = C\langle t \rangle_G$   
 $\langle proof \rangle$

**lemma** *ctxt-of-gctxt-apply-gterm*:  
 $gterm-of-term\ (ctxt-of-gctxt\ C)\langle t \rangle = C\langle gterm-of-term\ t \rangle_G$   
 $\langle proof \rangle$

**lemma** *ground-gctxt-of-ctxt-apply-gterm*:  
**assumes** *ground-ctxt*  $C$   
**shows**  $term-of-gterm\ (gctxt-of-ctxt\ C)\langle t \rangle_G = C\langle term-of-gterm\ t \rangle$   $\langle proof \rangle$

**lemma** *ground-gctxt-of-ctxt-apply* [*simp*]:  
**assumes** *ground-ctxt*  $C$  *ground*  $t$   
**shows**  $term-of-gterm\ (gctxt-of-ctxt\ C)\langle gterm-of-term\ t \rangle_G = C\langle t \rangle$   $\langle proof \rangle$

**lemma** *term-of-gterm-ctxt-apply* [simp]:  
 $term-of-gterm\ s = C\langle l \rangle \implies (gctxt-of-ctxt\ C)\langle gterm-of-term\ l \rangle_G = s$   
 ⟨proof⟩

**lemma** *gctxt-apply-inj-term*: *inj* (gctxt-apply-term C)  
 ⟨proof⟩

**lemma** *gctxt-apply-inj-on-term*: *inj-on* (gctxt-apply-term C) S  
 ⟨proof⟩

**lemma** *ctxt-of-pos-gterm* [simp]:  
 $p \in gposs\ t \implies ctxt-of-pos-term\ p\ (term-of-gterm\ t) = ctxt-of-gctxt\ (gctxt-at-pos\ t\ p)$   
 ⟨proof⟩

**lemma** *gctxt-of-gpos-gterm-gsubt-at-to-gterm* [simp]:  
 assumes  $p \in gposs\ t$   
 shows  $(gctxt-at-pos\ t\ p)\langle gsubt-at\ t\ p \rangle_G = t$  ⟨proof⟩

The position of the hole in a context is uniquely determined

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

**lemma** *ghole-pos-gctxt-at-pos* [simp]:  
 $p \in gposs\ t \implies ghole-pos\ (gctxt-at-pos\ t\ p) = p$   
 ⟨proof⟩

**lemma** *ghole-pos-id-ctxt* [simp]:  
 $C\langle s \rangle_G = t \implies gctxt-at-pos\ t\ (ghole-pos\ C) = C$   
 ⟨proof⟩

**lemma** *ghole-pos-in-apply*:  
 $ghole-pos\ C = p \implies p \in gposs\ C\langle u \rangle_G$   
 ⟨proof⟩

**lemma** *ground-hole-pos-to-ghole*:  
 $ground-ctxt\ C \implies ghole-pos\ (gctxt-of-ctxt\ C) = hole-pos\ C$   
 ⟨proof⟩

**lemma** *gsubst-at-gctxt-at-eq-gtermD*:  
 assumes  $s = t\ p \in gposs\ t$   
 shows  $gsubst-at\ s\ p = gsubst-at\ t\ p \wedge gctxt-at-pos\ s\ p = gctxt-at-pos\ t\ p$  ⟨proof⟩

**lemma** *gsubst-at-gctxt-at-eq-gtermI*:  
 assumes  $p \in gposs\ s\ p \in gposs\ t$   
 and  $gsubst-at\ s\ p = gsubst-at\ t\ p$   
 and  $gctxt-at-pos\ s\ p = gctxt-at-pos\ t\ p$

**shows**  $s = t$   $\langle proof \rangle$

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

$gsubt\text{-at } C \langle u \rangle_G (ghole\text{-pos } C) = u$   
 $\langle proof \rangle$

**lemma** *gctxt-at-pos-gsubt-at-pos* [simp]:

$p \in gposs\ t \implies gsubt\text{-at } (gctxt\text{-at-pos } t\ p) \langle u \rangle_G\ p = u$   
 $\langle proof \rangle$

**lemma** *gfun-at-gctxt-at-pos-not-after*:

**assumes**  $p \in gposs\ t\ q \in gposs\ t \neg (p \leq_p q)$   
**shows**  $gfun\text{-at } (gctxt\text{-at-pos } t\ p) \langle v \rangle_G\ q = gfun\text{-at } t\ q$   $\langle proof \rangle$

**lemma** *gposs-ConsE* [elim]:

**assumes**  $i \# p \in gposs\ t$   
**obtains**  $f\ ts$  **where**  $t = GFun\ f\ ts\ ts \neq []\ i < length\ ts\ p \in gposs\ (ts\ !\ i)$   $\langle proof \rangle$

**lemma** *gposs-gctxt-at-pos-not-after*:

**assumes**  $p \in gposs\ t\ q \in gposs\ t \neg (p \leq_p q)$   
**shows**  $q \in gposs\ (gctxt\text{-at-pos } t\ p) \langle v \rangle_G \longleftrightarrow q \in gposs\ t$   $\langle proof \rangle$

**lemma** *gposs-gctxt-at-pos*:

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

**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**  $gctxt\text{-at-pos } s\ p = gctxt\text{-at-pos } t\ p$   $\langle proof \rangle$

Signature of a ground context

**fun** *funas-gctxt* ::  $'f\ gctxt \Rightarrow ('f \times nat)$  **set where**

$funas\text{-gctxt } GHole = \{\}$  |  
 $funas\text{-gctxt } (GMore\ f\ ss1\ D\ ss2) = \{(f,\ Suc\ (length\ (ss1\ @\ ss2)))\}$   
 $\cup\ funas\text{-gctxt } D \cup \bigcup (set\ (map\ funas\text{-gterm } (ss1\ @\ ss2)))$

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

$ground\text{-ctxt } C \implies funas\text{-gctxt } (gctxt\text{-of-ctxt } C) = funas\text{-ctxt } C$   
 $\langle proof \rangle$

**lemma** *funas-ctxt-of-gctxt-conv* [simp]:

$funas\text{-ctxt } (ctxt\text{-of-gctxt } C) = funas\text{-gctxt } C$   
 $\langle proof \rangle$

**lemma** *inj-gctxt-of-ctxt-on-ground*:

*inj-on gtxt-of-ctxt (Collect ground-ctxt)*  
 ⟨proof⟩

**lemma** *funas-gterm-ctxt-apply* [simp]:  
*funas-gterm*  $C\langle s \rangle_G = \text{funas-gtxt } C \cup \text{funas-gterm } s$   
 ⟨proof⟩

**lemma** *funas-gtxt-compose* [simp]:  
*funas-gtxt*  $(C \circ_{Gc} D) = \text{funas-gtxt } C \cup \text{funas-gtxt } D$   
 ⟨proof⟩

**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* ::  $(f \times \text{nat}) \text{ set} \Rightarrow 'f \text{ gterm rel} \Rightarrow 'f \text{ gterm rel}$  **for**  $\mathcal{F} \mathcal{R}$   
**where**

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

**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⟩

**lemma** *gmctxt-cl-refl*:  
 $\text{funas-gterm } t \subseteq \mathcal{F} \Longrightarrow (t, t) \in \text{gmctxt-cl } \mathcal{F} \mathcal{R}$   
 ⟨proof⟩

**lemma** *gmctxt-cl-swap*:  
 $\text{gmctxt-cl } \mathcal{F} (\text{prod.swap } \mathcal{R}) = \text{prod.swap } \mathcal{R} (\text{gmctxt-cl } \mathcal{F} \mathcal{R})$  (**is**  $?Ls = ?Rs$ )  
 ⟨proof⟩

**lemma** *gmctxt-cl-mono-funas*:  
**assumes**  $\mathcal{F} \subseteq \mathcal{G}$  **shows**  $\text{gmctxt-cl } \mathcal{F} \mathcal{R} \subseteq \text{gmctxt-cl } \mathcal{G} \mathcal{R}$   
 ⟨proof⟩

**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⟩

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

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

**lemma**  $gcomp\text{-}rel$ :

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

### 2.6.5 Signature closed property

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

**lemma**  $all\text{-}ctxt\text{-}closed\text{-}gtermI$ :

**assumes**  $\bigwedge f \ ss \ ts. (f, length \ ss) \in \mathcal{F} \implies length \ ss = length \ ts \implies$   
 $(\forall i < length \ ts. (ss ! i, ts ! i) \in r) \implies (GFun \ f \ ss, GFun \ f \ ts) \in r$   
**shows**  $all\text{-}ctxt\text{-}closed\text{-}gterm \mathcal{F} r$   $\langle proof \rangle$

**lemma**  $all\text{-}ctxt\text{-}closed\text{-}gtermD$ :

$all\text{-}ctxt\text{-}closed\text{-}gterm F r \implies (f, length \ ss) \in F \implies length \ ss = length \ ts \implies$   
 $(\forall i < length \ ts. (ss ! i, ts ! i) \in r) \implies (GFun \ f \ ss, GFun \ f \ ts) \in r$   
 $\langle proof \rangle$

**lemma**  $all\text{-}ctxt\text{-}closed\text{-}gterm\text{-}refl\text{-}on$ :

**assumes**  $all\text{-}ctxt\text{-}closed\text{-}gterm \mathcal{F} r \ s \in \mathcal{T}_G \mathcal{F}$   
**shows**  $(s, s) \in r$   $\langle proof \rangle$

**lemma**  $gmctxt\text{-}cl\text{-}is\text{-}all\text{-}ctxt\text{-}closed\text{-}gterm$  [simp]:

$all\text{-}ctxt\text{-}closed\text{-}gterm \mathcal{F} (gmctxt\text{-}cl \mathcal{F} \mathcal{R})$   
 $\langle proof \rangle$

**lemma**  $all\text{-}ctxt\text{-}closed\text{-}gterm\text{-}gmctxt\text{-}cl\text{-}idem$  [simp]:

**assumes**  $all\text{-}ctxt\text{-}closed\text{-}gterm \mathcal{F} \mathcal{R}$   
**shows**  $gmctxt\text{-}cl \mathcal{F} \mathcal{R} = \mathcal{R}$   
 $\langle proof \rangle$

### 2.6.6 Transitive closure preserves $all\text{-}ctxt\text{-}closed\text{-}gterm$

induction scheme for transitive closures of lists

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

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

**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>

**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>

**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>

**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**  $P \ ss \ ts$  <proof>

**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**  $P \ ss \ ts$  <proof>

**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>

**end**

**theory** *Horn-Inference*

**imports** *Main*

**begin**

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

**locale** *horn =*

**fixes**  $\mathcal{H} :: 'a \ \text{horn} \ \text{set}$

**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*  $x B = \{a \mid as \ a. as \rightarrow_h a \in \mathcal{H} \wedge x \in \text{set } as \wedge \text{set } as \subseteq B \cup \{x\}\}$

**inductive step** :: 'a set  $\times$  'a set  $\Rightarrow$  'a set  $\times$  'a set  $\Rightarrow$  bool (**infix**  $\langle \vdash \rangle$  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**:

$(G, B) \vdash (G', B') \implies (H \cup G, B) \vdash (H \cup G', B')$

*<proof>*

**fun invariant** where

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

**lemma inv-start**:

**shows** *invariant* (*infer0*,  $\{\}$ )

*<proof>*

**lemma inv-step**:

**assumes** *invariant*  $(G, B)$   $(G, B) \vdash (G', B')$

**shows** *invariant*  $(G', B')$

*<proof>*

**lemma inv-end**:

**assumes** *invariant*  $(\{\}, B)$

**shows**  $B = \text{saturate}$

*<proof>*

**lemma step-sound**:

$(\text{infer0}, \{\}) \vdash (\{\}, B) \implies B = \text{saturate}$

*<proof>*

**end**

**lemma horn-infer0-union**:

*horn.infer0*  $(\mathcal{H}_1 \cup \mathcal{H}_2) = \text{horn.infer0 } \mathcal{H}_1 \cup \text{horn.infer0 } \mathcal{H}_2$

*<proof>*

**lemma horn-infer1-union**:

*horn.infer1*  $(\mathcal{H}_1 \cup \mathcal{H}_2) \ x \ B = \text{horn.infer1 } \mathcal{H}_1 \ x \ B \cup \text{horn.infer1 } \mathcal{H}_2 \ x \ B$

```

    <proof>

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 (λxa. case-option None (f xa)) xs None = None
  <proof>

lemma saturate-fold-mono [partial-function-mono]:
  option.mono-body (λf. fold (λx. case-option None (λy. f (x, y))) xs b)
  <proof>

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>

lemma saturate-impl-sound:
  assumes saturate-impl = Some B'
  shows set B' = saturate
  <proof>

lemma saturate-impl-complete:
  assumes finite saturate
  shows saturate-impl ≠ None
  <proof>

```

**end**

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

**end**

**theory** Horn-Fset

**imports** Horn-Inference FSet-Utils

**begin**

**locale** horn-fset-impl = horn +

**fixes** infer0-impl :: 'a list **and** infer1-impl :: 'a  $\Rightarrow$  'a fset  $\Rightarrow$  'a list

**begin**

**lemma** saturate-fold-simp [simp]:

*fold* ( $\lambda xa. \text{case-option None } (f \ x a)$ ) *xs None = None*

*<proof>*

**lemma** saturate-fold-mono [partial-function-mono]:

*option.mono-body* ( $\lambda f. \text{fold } (\lambda x. \text{case-option None } (\lambda y. f \ (x, y))) \ x \ b$ )

*<proof>*

**partial-function** (*option*) saturate-rec :: 'a  $\Rightarrow$  'a fset  $\Rightarrow$  ('a fset) **option** **where**

*saturate-rec* *x bs = (if* *x* *| $\in$ |* *bs* *then* *Some* *bs* *else*

*fold* ( $\lambda x. \text{case-option None } (\text{saturate-rec } x)$ ) (*infer1-impl* *x bs*) (*Some* (*finsert* *x* *bs*)))

**definition** saturate-impl **where**

*saturate-impl = fold* ( $\lambda x. \text{case-option None } (\text{saturate-rec } x)$ ) *infer0-impl* (*Some*  $\{\|\}$ )

**end**

**locale** horn-fset = horn-fset-impl +

**assumes** infer0: *infer0 = set infer0-impl*

**and** infer1:  $\bigwedge x \ bs. \text{infer1 } x \ (\text{fset } bs) = \text{set } (\text{infer1-impl } x \ bs)$

**begin**

**lemma** saturate-rec-sound:

*saturate-rec* *x bs = Some* *bs'*  $\Longrightarrow$  ( $\{x\}, \text{fset } bs$ )  $\vdash$  ( $\{\}, \text{fset } bs'$ )

*<proof>*

**lemma** saturate-impl-sound:

**assumes** saturate-impl = *Some* *B'*

**shows** *fset* *B' = saturate*

*<proof>*

**lemma** saturate-impl-complete:

**assumes** *finite* *saturate*

**shows** *saturate-impl*  $\neq$  *None*

*<proof>*

**end**

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

**end**

### 3 Tree automaton

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

#### 3.1 Tree automaton definition and functionality

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

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

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

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

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

**abbreviation** *rule-arg-states* **where** *rule-arg-states*  $\Delta \equiv |\cup| ((fset-of-list \circ r-lhs-states) |^{\dagger} \Delta)$

**abbreviation** *rule-target-states* **where** *rule-target-states*  $\Delta \equiv (r-rhs |^{\dagger} \Delta)$

**definition** *rule-states* **where** *rule-states*  $\Delta \equiv rule-arg-states \Delta | \cup | rule-target-states \Delta$

**definition** *eps-states* **where** *eps-states*  $\Delta_{\varepsilon} \equiv (fst |^{\dagger} \Delta_{\varepsilon}) | \cup | (snd |^{\dagger} \Delta_{\varepsilon})$

**definition**  $\mathcal{Q} \mathcal{A} = rule-states (rules \mathcal{A}) | \cup | eps-states (eps \mathcal{A})$

**abbreviation**  $\mathcal{Q}_r \mathcal{A} \equiv \mathcal{Q} (ta \mathcal{A})$

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

*ta-rhs-states*  $\mathcal{A} \equiv \{ | q | p q. (p | \in | rule-target-states (rules \mathcal{A})) \wedge (p = q \vee (p, q) | \in | (eps \mathcal{A})^{+} |) | \}$

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

##### 3.1.1 Rechability of a term induced by a tree automaton

**fun** *ta-der* :: ('q, 'f) *ta*  $\Rightarrow$  ('f, 'q) *term*  $\Rightarrow$  'q fset **where**

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

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

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

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

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

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

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

### 3.1.2 Language acceptance

**definition**  $ta\text{-}lang :: 'q fset \Rightarrow ('q, 'f) ta \Rightarrow ('f, 'v) terms$  **where**

$[code del]: ta\text{-}lang Q \mathcal{A} = \{adapt\text{-}vars t \mid t . ground t \wedge Q \mid \cap | ta\text{-}der \mathcal{A} t \neq \{\}\}$

**definition**  $gta\text{-}der$  **where**

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

**definition**  $gta\text{-}lang$  **where**

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

**definition**  $\mathcal{L}$  **where**

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

**definition**  $reg\text{-}Restr\text{-}Q_f$  **where**

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

### 3.1.3 Trimming

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

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

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

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

**definition**  $ta\text{-productive} :: 'q \text{ fset} \Rightarrow ('q, 'f) ta \Rightarrow 'q \text{ fset}$  **where**  
 $ta\text{-productive } P \mathcal{A} \equiv \{ |q| q q' C. q' \in | ta\text{-der } \mathcal{A} (C \langle \text{Var } q \rangle) \wedge q' \in | P | \}$

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

**definition**  $ta\text{-is-trim} :: 'q \text{ fset} \Rightarrow ('q, 'f) ta \Rightarrow \text{bool}$  **where**  
 $ta\text{-is-trim } P \mathcal{A} \equiv \forall q. q \in | \mathcal{Q} \mathcal{A} \longrightarrow q \in | ta\text{-reachable } \mathcal{A} \wedge q \in | ta\text{-productive } P \mathcal{A} \}$

**definition**  $reg\text{-is-trim} :: ('q, 'f) reg \Rightarrow \text{bool}$  **where**  
 $reg\text{-is-trim } R \equiv ta\text{-is-trim } (fin R) (ta R)$

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

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

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

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

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

**definition**  $trim\text{-ta} :: 'q \text{ fset} \Rightarrow ('q, 'f) ta \Rightarrow ('q, 'f) ta$  **where**  
 $trim\text{-ta } P \mathcal{A} = ta\text{-only-prod } P (ta\text{-only-reach } \mathcal{A})$

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

### 3.1.4 Mapping over tree automata

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

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

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

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

### 3.1.5 Product construction (language intersection)

**definition** *prod-ta-rules* :: ('q1,'f) ta ⇒ ('q2,'f) ta ⇒ ('q1 × 'q2, 'f) ta-rule fset  
**where**

*prod-ta-rules*  $\mathcal{A}$   $\mathcal{B}$  = { | TA-rule f qs q | f qs q. TA-rule f (map fst qs) (fst q) | ∈ |  
*rules*  $\mathcal{A}$  ∧

TA-rule f (map snd qs) (snd q) | ∈ | *rules*  $\mathcal{B}$  }

**declare** *prod-ta-rules-def* [simp]

**definition** *prod-epsLp* **where**

*prod-epsLp*  $\mathcal{A}$   $\mathcal{B}$  = (λ (p, q). (fst p, fst q) | ∈ | *eps*  $\mathcal{A}$  ∧ snd p = snd q ∧ snd q | ∈ |  
 $\mathcal{Q}$   $\mathcal{B}$ )

**definition** *prod-epsRp* **where**

*prod-epsRp*  $\mathcal{A}$   $\mathcal{B}$  = (λ (p, q). (snd p, snd q) | ∈ | *eps*  $\mathcal{B}$  ∧ fst p = fst q ∧ fst q | ∈ |  
 $\mathcal{Q}$   $\mathcal{A}$ )

**definition** *prod-ta* :: ('q1,'f) ta ⇒ ('q2,'f) ta ⇒ ('q1 × 'q2, 'f) ta **where**

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

(fCollect (*prod-epsLp*  $\mathcal{A}$   $\mathcal{B}$ ) | ∪ | fCollect (*prod-epsRp*  $\mathcal{A}$   $\mathcal{B}$ ))

**definition** *reg-intersect* **where**

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

### 3.1.6 Union construction (language union)

**definition** *ta-union* **where**

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

**definition** *reg-union* **where**

*reg-union* R L = Reg (Inl | ∣ (fin R | ∩ |  $\mathcal{Q}_r$  R) | ∪ | Inr | ∣ (fin L | ∩ |  $\mathcal{Q}_r$  L))

(*ta-union* (fmap-states-ta Inl (ta R)) (fmap-states-ta Inr (ta L)))

### 3.1.7 Epsilon free and tree automaton accepting empty language

**definition** *eps-free-rulep* **where**

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

**definition** *eps-free* :: ('q, 'f) ta ⇒ ('q, 'f) ta **where**

*eps-free*  $\mathcal{A}$  = TA (fCollect (*eps-free-rulep*  $\mathcal{A}$ )) {||}

**definition** *is-ta-eps-free* :: ('q, 'f) ta ⇒ bool **where**

*is-ta-eps-free*  $\mathcal{A}$  ↔ *eps*  $\mathcal{A}$  = {||}

**definition** *ta-empty* :: 'q fset ⇒ ('q,'f) ta ⇒ bool **where**

*ta-empty* Q  $\mathcal{A}$  ↔ *ta-reachable*  $\mathcal{A}$  | ∩ | Q | ⊆ | {||}

**definition** *eps-free-reg* **where**

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

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

### 3.1.8 Relabeling tree automaton states to natural numbers

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

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

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

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

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

— The instantiation of  $<$  and  $\leq$  for finite sets are  $|\subset|$  and  $|\subseteq|$  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-fset-ta*  $:: (('q :: \text{linorder}) \text{fset}, 'f) \text{ta} \Rightarrow (\text{nat}, 'f) \text{ta}$  **where**  
*relabel-fset-ta*  $\mathcal{A} = \text{fmap-states-ta } (\text{map-fset-fset-to-nat } (\mathcal{Q} \mathcal{A})) \mathcal{A}$

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

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

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

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

**lemma** *rules-transfer* [*transfer-rule*]:  
*rel-fun*  $(=) (\text{pcr-fset } (=)) \text{srules } \text{rules} \langle \text{proof} \rangle$

**lemma** *eps-transfer* [*transfer-rule*]:  
*rel-fun*  $(=) (\text{pcr-fset } (=)) \text{seps } \text{eps} \langle \text{proof} \rangle$

**lemma** *TA-equalityI*:

$rules\ \mathcal{A} = rules\ \mathcal{B} \implies eps\ \mathcal{A} = eps\ \mathcal{B} \implies \mathcal{A} = \mathcal{B}$   
 ⟨proof⟩

**lemma** *rule-states-code* [code]:  
 $rule\text{-}states\ \Delta = |\bigcup| ((\lambda\ r.\ finsert\ (r\text{-}rhs\ r)\ (fset\text{-}of\text{-}list\ (r\text{-}lhs\text{-}states\ r))))\ |\uparrow\ \Delta$   
 ⟨proof⟩

**lemma** *eps-states-code* [code]:  
 $eps\text{-}states\ \Delta_\varepsilon = |\bigcup| ((\lambda\ (q, q').\ \{|q, q'|\})\ |\uparrow\ \Delta_\varepsilon)\ (\mathbf{is}\ ?Ls = ?Rs)$   
 ⟨proof⟩

**lemma** *rule-states-empty* [simp]:  
 $rule\text{-}states\ \{\|\} = \{\|\}$   
 ⟨proof⟩

**lemma** *eps-states-empty* [simp]:  
 $eps\text{-}states\ \{\|\} = \{\|\}$   
 ⟨proof⟩

**lemma** *rule-states-union* [simp]:  
 $rule\text{-}states\ (\Delta\ |\cup|\ \Gamma) = rule\text{-}states\ \Delta\ |\cup|\ rule\text{-}states\ \Gamma$   
 ⟨proof⟩

**lemma** *rule-states-mono*:  
 $\Delta\ |\subseteq|\ \Gamma \implies rule\text{-}states\ \Delta\ |\subseteq|\ rule\text{-}states\ \Gamma$   
 ⟨proof⟩

**lemma** *eps-states-union* [simp]:  
 $eps\text{-}states\ (\Delta\ |\cup|\ \Gamma) = eps\text{-}states\ \Delta\ |\cup|\ eps\text{-}states\ \Gamma$   
 ⟨proof⟩

**lemma** *eps-states-image* [simp]:  
 $eps\text{-}states\ (map\text{-}both\ f\ |\uparrow\ \Delta_\varepsilon) = f\ |\uparrow\ eps\text{-}states\ \Delta_\varepsilon$   
 ⟨proof⟩

**lemma** *eps-states-mono*:  
 $\Delta\ |\subseteq|\ \Gamma \implies eps\text{-}states\ \Delta\ |\subseteq|\ eps\text{-}states\ \Gamma$   
 ⟨proof⟩

**lemma** *eps-statesI* [intro]:  
 $(p, q)\ |\in|\ \Delta \implies p\ |\in|\ eps\text{-}states\ \Delta$   
 $(p, q)\ |\in|\ \Delta \implies q\ |\in|\ eps\text{-}states\ \Delta$   
 ⟨proof⟩

**lemma** *eps-statesE* [elim]:  
 assumes  $p\ |\in|\ eps\text{-}states\ \Delta$   
 obtains  $q$  where  $(p, q)\ |\in|\ \Delta \vee (q, p)\ |\in|\ \Delta$  ⟨proof⟩

**lemma** *rule-statesE* [elim]:

**assumes**  $q \in | \text{rule-states } \Delta$   
**obtains**  $f \text{ ps } p$  **where** *TA-rule*  $f \text{ ps } p \in | \Delta \ q \in | (\text{fset-of-list } \text{ps}) \vee q = p$  *<proof>*

**lemma** *rule-statesI* [*intro*]:  
**assumes**  $r \in | \Delta \ q \in | \text{finsert } (r\text{-rhs } r) (\text{fset-of-list } (r\text{-lhs-states } r))$   
**shows**  $q \in | \text{rule-states } \Delta$  *<proof>*

Destruction rule for states

**lemma** *rule-statesD*:  
 $r \in | (\text{rules } \mathcal{A}) \implies r\text{-rhs } r \in | \mathcal{Q} \ \mathcal{A} \ f \ \text{qs} \rightarrow q \in | (\text{rules } \mathcal{A}) \implies q \in | \mathcal{Q} \ \mathcal{A}$   
 $r \in | (\text{rules } \mathcal{A}) \implies p \in | \text{fset-of-list } (r\text{-lhs-states } r) \implies p \in | \mathcal{Q} \ \mathcal{A}$   
 $f \ \text{qs} \rightarrow q \in | (\text{rules } \mathcal{A}) \implies p \in | \text{fset-of-list } \text{qs} \implies p \in | \mathcal{Q} \ \mathcal{A}$   
*<proof>*

**lemma** *eps-states* [*simp*]:  $(\text{eps } \mathcal{A}) \subseteq | \mathcal{Q} \ \mathcal{A} \ |\times| \ \mathcal{Q} \ \mathcal{A}$   
*<proof>*

**lemma** *eps-statesD*:  $(p, q) \in | (\text{eps } \mathcal{A}) \implies p \in | \mathcal{Q} \ \mathcal{A} \wedge q \in | \mathcal{Q} \ \mathcal{A}$   
*<proof>*

**lemma** *eps-trancl-statesD*:  
 $(p, q) \in | (\text{eps } \mathcal{A})^+ \implies p \in | \mathcal{Q} \ \mathcal{A} \wedge q \in | \mathcal{Q} \ \mathcal{A}$   
*<proof>*

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

Mapping over function symbols/states

**lemma** *finite-Collect-ta-rule*:  
 $\text{finite } \{ \text{TA-rule } f \ \text{qs } q \mid f \ \text{qs } q. \text{ TA-rule } f \ \text{qs } q \in | \text{rules } \mathcal{A} \}$  (**is finite** ?*S*)  
*<proof>*

**lemma** *map-ta-rule-finite*:  
 $\text{finite } \Delta \implies \text{finite } \{ \text{TA-rule } (g \ h) (\text{map } f \ \text{qs}) (f \ q) \mid h \ \text{qs } q. \text{ TA-rule } h \ \text{qs } q \in \Delta \}$   
*<proof>*

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

**lemmas** *map-ta-rule-states-finite* [*simp*] = *map-ta-rule-finite*[*of fset*  $\Delta$  *id* **for**  $\Delta$ , *simplified*]

**lemmas** *map-ta-rule-funsym-finite* [*simp*] = *map-ta-rule-finite*[*of fset*  $\Delta$  - *id* **for**  $\Delta$ , *simplified*]

**lemma** *map-ta-rule-comp*:  
 $\text{map-ta-rule } f \ g \circ \text{map-ta-rule } f' \ g' = \text{map-ta-rule } (f \circ f') (g \circ g')$   
*<proof>*

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

**lemma** *map-ta-rule-prod-swap-id* [*simp*]:  
 $map-ta-rule\ prod.swap\ prod.swap\ (map-ta-rule\ prod.swap\ prod.swap\ r) = r$   
 $\langle proof \rangle$

**lemma** *rule-states-image* [*simp*]:  
 $rule-states\ (map-ta-rule\ f\ g\ |^q\ \Delta) = f\ |^q\ rule-states\ \Delta$  (**is**  $?Ls = ?Rs$ )  
 $\langle proof \rangle$

**lemma** *Q-mono*:  
 $(rules\ \mathcal{A})\ |\subseteq|\ (rules\ \mathcal{B}) \implies (eps\ \mathcal{A})\ |\subseteq|\ (eps\ \mathcal{B}) \implies \mathcal{Q}\ \mathcal{A}\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$   
 $\langle proof \rangle$

**lemma** *Q-subseteq-I*:  
**assumes**  $\bigwedge r. r\ |\in|\ rules\ \mathcal{A} \implies r-rhs\ r\ |\in|\ S$   
**and**  $\bigwedge r. r\ |\in|\ rules\ \mathcal{A} \implies fset-of-list\ (r-lhs-states\ r)\ |\subseteq|\ S$   
**and**  $\bigwedge e. e\ |\in|\ eps\ \mathcal{A} \implies fst\ e\ |\in|\ S \wedge snd\ e\ |\in|\ S$   
**shows**  $\mathcal{Q}\ \mathcal{A}\ |\subseteq|\ S$   $\langle proof \rangle$

**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**  
 $finite\ ?set$ )  
 $\langle proof \rangle$

Collecting all states reachable from target of rules

**lemma** *finite-ta-rhs-states* [*simp*]:  
 $finite\ \{q. \exists p. p\ |\in|\ rule-target-states\ (rules\ \mathcal{A}) \wedge (p = q \vee (p, q)\ |\in|\ (eps\ \mathcal{A})^{+})\}$   
(**is**  $finite\ ?Set$ )  
 $\langle proof \rangle$

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

**lemma** *ta-sigI* [*intro*]:  
 $TA-rule\ f\ qs\ q\ |\in|\ (rules\ \mathcal{A}) \implies length\ qs = n \implies (f, n)\ |\in|\ ta-sig\ \mathcal{A}$   $\langle proof \rangle$

**lemma** *ta-sig-mono*:  
 $(rules\ \mathcal{A})\ |\subseteq|\ (rules\ \mathcal{B}) \implies ta-sig\ \mathcal{A}\ |\subseteq|\ ta-sig\ \mathcal{B}$   
 $\langle proof \rangle$

**lemma** *finite-eps*:  
 $finite\ \{q. \exists f\ ps\ p. f\ ps \rightarrow p\ |\in|\ rules\ \mathcal{A} \wedge (p = q \vee (p, q)\ |\in|\ (eps\ \mathcal{A})^{+})\}$  (**is**  
 $finite\ ?S$ )  
 $\langle proof \rangle$

**lemma** *collect-snd-trancl-fset*:  
 $\{p. (q, p)\ |\in|\ (eps\ \mathcal{A})^{+}\} = fset\ (snd\ |^q\ (ffilter\ (\lambda x. fst\ x = q)\ ((eps\ \mathcal{A})^{+})))$   
 $\langle proof \rangle$

**lemma** *ta-der-Var*:

$q \mid \in \mid ta\text{-der } \mathcal{A} (Var x) \longleftrightarrow x = q \vee (x, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid$   
 $\langle proof \rangle$

**lemma** *ta-der-Fun*:

$q \mid \in \mid ta\text{-der } \mathcal{A} (Fun f ts) \longleftrightarrow (\exists ps p. TA\text{-rule } f ps p \mid \in \mid (rules \mathcal{A}) \wedge$   
 $(p = q \vee (p, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid) \wedge length ps = length ts \wedge$   
 $(\forall i < length ts. ps ! i \mid \in \mid ta\text{-der } \mathcal{A} (ts ! i)))$  (**is**  $?Ls \longleftrightarrow ?Rs$ )  
 $\langle proof \rangle$

**declare** *ta-der.simps*[*simp del*]

**declare** *ta-der.simps*[*code del*]

**lemmas** *ta-der-simps* [*simp*] = *ta-der-Var ta-der-Fun*

**lemma** *ta-der'-Var*:

$Var q \mid \in \mid ta\text{-der}' \mathcal{A} (Var x) \longleftrightarrow x = q \vee (x, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid$   
 $\langle proof \rangle$

**lemma** *ta-der'-Fun*:

$Var q \mid \in \mid ta\text{-der}' \mathcal{A} (Fun f ts) \longleftrightarrow q \mid \in \mid ta\text{-der } \mathcal{A} (Fun f ts)$   
 $\langle proof \rangle$

**lemma** *ta-der'-Fun2*:

$Fun f ps \mid \in \mid ta\text{-der}' \mathcal{A} (Fun g ts) \longleftrightarrow f = g \wedge length ps = length ts \wedge (\forall i < length$   
 $ts. ps ! i \mid \in \mid ta\text{-der}' \mathcal{A} (ts ! i))$   
 $\langle proof \rangle$

**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*:  $q \mid \in \mid ta\text{-der } \mathcal{A} t$   
**and** *VarI*:  $\bigwedge q v. v = q \vee (v, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid \implies P (Var v) q$   
**and** *FunI*:  $\bigwedge f ts ps p q. f ps \rightarrow p \mid \in \mid rules \mathcal{A} \implies length ts = length ps \implies p =$   
 $q \vee (p, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid \implies$   
 $(\bigwedge i. i < length ts \implies ps ! i \mid \in \mid ta\text{-der } \mathcal{A} (ts ! i)) \implies$   
 $(\bigwedge i. i < length ts \implies P (ts ! i) (ps ! i)) \implies P (Fun f ts) q$   
**shows**  $P t q$   $\langle proof \rangle$

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

**assumes** *reach*:  $q \mid \in \mid ta\text{-der } \mathcal{A} (term\text{-of-gterm } t)$   
**and** *Fun*:  $\bigwedge f ts ps p q. TA\text{-rule } f ps p \mid \in \mid rules \mathcal{A} \implies length ts = length ps \implies$   
 $p = q \vee (p, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \mid \implies$   
 $(\bigwedge i. i < length ts \implies ps ! i \mid \in \mid ta\text{-der } \mathcal{A} (term\text{-of-gterm } (ts ! i))) \implies$   
 $(\bigwedge i. i < length ts \implies P (ts ! i) (ps ! i)) \implies P (GFun f ts) q$   
**shows**  $P t q$   $\langle proof \rangle$

**lemma** *ta-der-rule-empty*:

**assumes**  $q \in | \text{ta-der} (TA \{|\} \Delta_\varepsilon) t$   
**obtains**  $p$  **where**  $t = \text{Var } p \ p = q \vee (p, q) \in | \Delta_\varepsilon|^+ |$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-eps*:

**assumes**  $(p, q) \in | (\text{eps } \mathcal{A})$  **and**  $p \in | \text{ta-der } \mathcal{A} t$   
**shows**  $q \in | \text{ta-der } \mathcal{A} t \langle \text{proof} \rangle$

**lemma** *ta-der-trancl-eps*:

**assumes**  $(p, q) \in | (\text{eps } \mathcal{A})^+ |$  **and**  $p \in | \text{ta-der } \mathcal{A} t$   
**shows**  $q \in | \text{ta-der } \mathcal{A} t \langle \text{proof} \rangle$

**lemma** *ta-der-mono*:

$(\text{rules } \mathcal{A}) \subseteq | (\text{rules } \mathcal{B}) \implies (\text{eps } \mathcal{A}) \subseteq | (\text{eps } \mathcal{B}) \implies \text{ta-der } \mathcal{A} t \subseteq | \text{ta-der } \mathcal{B} t$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-el-mono*:

$(\text{rules } \mathcal{A}) \subseteq | (\text{rules } \mathcal{B}) \implies (\text{eps } \mathcal{A}) \subseteq | (\text{eps } \mathcal{B}) \implies q \in | \text{ta-der } \mathcal{A} t \implies q \in | \text{ta-der } \mathcal{B} t$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der'-ta-der*:

**assumes**  $t \in | \text{ta-der}' \mathcal{A} s \ p \in | \text{ta-der } \mathcal{A} t$   
**shows**  $p \in | \text{ta-der } \mathcal{A} s \langle \text{proof} \rangle$

**lemma** *ta-der'-empty*:

**assumes**  $t \in | \text{ta-der}' (TA \{|\} \{|\}) s$   
**shows**  $t = s \langle \text{proof} \rangle$

**lemma** *ta-der'-to-ta-der*:

$\text{Var } q \in | \text{ta-der}' \mathcal{A} s \implies q \in | \text{ta-der } \mathcal{A} s$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-to-ta-der'*:

$q \in | \text{ta-der } \mathcal{A} s \iff \text{Var } q \in | \text{ta-der}' \mathcal{A} s$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der'-poss*:

**assumes**  $t \in | \text{ta-der}' \mathcal{A} s$   
**shows**  $\text{poss } t \subseteq \text{poss } s \langle \text{proof} \rangle$

**lemma** *ta-der'-refl[simp]*:  $t \in | \text{ta-der}' \mathcal{A} t$

$\langle \text{proof} \rangle$

**lemma** *ta-der'-eps*:

**assumes**  $\text{Var } p \in | \text{ta-der}' \mathcal{A} s$  **and**  $(p, q) \in | (\text{eps } \mathcal{A})^+ |$   
**shows**  $\text{Var } q \in | \text{ta-der}' \mathcal{A} s \langle \text{proof} \rangle$

**lemma** *ta-der'-trans*:

**assumes**  $t \in \text{ta-der}' \mathcal{A} s$  **and**  $u \in \text{ta-der}' \mathcal{A} t$   
**shows**  $u \in \text{ta-der}' \mathcal{A} s$   $\langle \text{proof} \rangle$

Connecting contexts to derivation definition

**lemma** *ta-der-ctxt*:

**assumes**  $p: p \in \text{ta-der} \mathcal{A} t$   $q \in \text{ta-der} \mathcal{A} C \langle \text{Var } p \rangle$   
**shows**  $q \in \text{ta-der} \mathcal{A} C \langle t \rangle$   $\langle \text{proof} \rangle$

**lemma** *ta-der-eps-ctxt*:

**assumes**  $p \in \text{ta-der} \mathcal{A} C \langle \text{Var } q \rangle$  **and**  $(q, q') \in (\text{eps } \mathcal{A})^+$   
**shows**  $p \in \text{ta-der} \mathcal{A} C \langle \text{Var } q \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *rule-reachable-ctxt-exist*:

**assumes** *rule*:  $f \text{ qs} \rightarrow q \in \text{rules } \mathcal{A}$  **and**  $i < \text{length } \text{qs}$   
**shows**  $\exists C. q \in \text{ta-der} \mathcal{A} (C \langle \text{Var } (\text{qs } ! i) \rangle)$   $\langle \text{proof} \rangle$

**lemma** *ta-der-ctxt-decompose*:

**assumes**  $q \in \text{ta-der} \mathcal{A} C \langle t \rangle$   
**shows**  $\exists p. p \in \text{ta-der} \mathcal{A} t \wedge q \in \text{ta-der} \mathcal{A} C \langle \text{Var } p \rangle$   $\langle \text{proof} \rangle$

**lemma** *ta-der-states*:

$\text{ta-der} \mathcal{A} t \subseteq \mathcal{Q} \mathcal{A} \cup \text{fvars-term } t$   
 $\langle \text{proof} \rangle$

**lemma** *ground-ta-der-states*:

$\text{ground } t \implies \text{ta-der} \mathcal{A} t \subseteq \mathcal{Q} \mathcal{A}$   
 $\langle \text{proof} \rangle$

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

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

$q \in \text{ta-der} \mathcal{A} (\text{term-of-gterm } t) \implies q \in \mathcal{Q} \mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-states'*:

$q \in \text{ta-der} \mathcal{A} t \implies q \in \mathcal{Q} \mathcal{A} \implies \text{fvars-term } t \subseteq \mathcal{Q} \mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-not-stateD*:

$q \in \text{ta-der} \mathcal{A} t \implies q \notin \mathcal{Q} \mathcal{A} \implies t = \text{Var } q$   
 $\langle \text{proof} \rangle$

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

$\text{is-Fun } t \implies q \in \text{ta-der} \mathcal{A} t \implies q \in \mathcal{Q} \mathcal{A}$   
 $\langle \text{proof} \rangle$

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

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

$\langle proof \rangle$

**lemma** *ta-der-not-reach*:

**assumes**  $\bigwedge r. r \in rules \mathcal{A} \implies r\text{-rhs } r \neq q$   
**and**  $\bigwedge e. e \in eps \mathcal{A} \implies snd e \neq q$   
**shows**  $q \notin ta\text{-der } \mathcal{A} (term\text{-of-gterm } t) \langle proof \rangle$

**lemma** *ta-rhs-states-subset-states*:  $ta\text{-rhs-states } \mathcal{A} \subseteq \mathcal{Q} \mathcal{A}$

$\langle proof \rangle$

**lemma** *ta-rhs-states-res*: **assumes** *is-Fun*  $t$

**shows**  $ta\text{-der } \mathcal{A} t \subseteq ta\text{-rhs-states } \mathcal{A}$

$\langle proof \rangle$

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

**lemma** *ta-der-adapt-vars-ground* [*simp*]:

$ground\ t \implies ta\text{-der } \mathcal{A} (adapt\text{-vars } t) = ta\text{-der } \mathcal{A} t$

$\langle proof \rangle$

**lemma** *gterm-of-term-inv*:

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

$\langle proof \rangle$

**lemma** *map-vars-term-term-of-gterm*:

$map\text{-vars-term } f (term\text{-of-gterm } t) = term\text{-of-gterm } t$

$\langle proof \rangle$

**lemma** *adapt-vars-term-of-gterm*:

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

$\langle proof \rangle$

**lemma** *ta-der-term-sig*:

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

$\langle proof \rangle$

**lemma** *ta-der-gterm-sig*:

$q \in ta\text{-der } \mathcal{A} (term\text{-of-gterm } t) \implies ffunas\text{-gterm } t \subseteq ta\text{-sig } \mathcal{A}$

$\langle proof \rangle$

*ta-lang* for terms with arbitrary variable type

**lemma** *ta-langE*: **assumes**  $t \in ta\text{-lang } \mathcal{Q} \mathcal{A}$

**obtains**  $t' q$  **where**  $ground\ t' q \in \mathcal{Q} q \in ta\text{-der } \mathcal{A} t' t = adapt\text{-vars } t'$

$\langle proof \rangle$

**lemma** *ta-langI*: **assumes**  $ground\ t' q \in \mathcal{Q} q \in ta\text{-der } \mathcal{A} t' t = adapt\text{-vars } t'$

**shows**  $t \in \text{ta-lang } Q \mathcal{A}$   
 ⟨proof⟩

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

*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⟩

**lemma** *term-of-gterm-in-ta-lang-conv*:  
 $\text{term-of-gterm } t \in \text{ta-lang } Q \mathcal{A} \longleftrightarrow t \in \text{gta-lang } Q \mathcal{A}$   
 ⟨proof⟩

**lemma** *gta-lang-def-sym*:  
 $\text{gterm-of-term } ' \text{ta-lang } Q \mathcal{A} = \text{gta-lang } Q \mathcal{A}$   
 ⟨proof⟩

**lemma** *gta-langI* [*intro*]:  
**assumes**  $q \mid \in \mid Q$  **and**  $q \mid \in \mid \text{ta-der } \mathcal{A}$  (*term-of-gterm*  $t$ )  
**shows**  $t \in \text{gta-lang } Q \mathcal{A}$  ⟨proof⟩

**lemma** *gta-langE* [*elim*]:  
**assumes**  $t \in \text{gta-lang } Q \mathcal{A}$   
**obtains**  $q$  **where**  $q \mid \in \mid Q$  **and**  $q \mid \in \mid \text{ta-der } \mathcal{A}$  (*term-of-gterm*  $t$ ) ⟨proof⟩

**lemma** *gta-lang-mono*:  
**assumes**  $\bigwedge t. \text{ta-der } \mathcal{A} t \mid \subseteq \mid \text{ta-der } \mathfrak{B} t$  **and**  $Q_{\mathcal{A}} \mid \subseteq \mid Q_{\mathfrak{B}}$   
**shows**  $\text{gta-lang } Q_{\mathcal{A}} \mathcal{A} \subseteq \text{gta-lang } Q_{\mathfrak{B}} \mathfrak{B}$   
 ⟨proof⟩

**lemma** *gta-lang-term-of-gterm* [*simp*]:  
 $\text{term-of-gterm } t \in \text{term-of-gterm } ' \text{gta-lang } Q \mathcal{A} \longleftrightarrow t \in \text{gta-lang } Q \mathcal{A}$   
 ⟨proof⟩

**lemma** *gta-lang-subset-rules-funas*:  
 $\text{gta-lang } Q \mathcal{A} \subseteq \mathcal{T}_G (\text{fset } (\text{ta-sig } \mathcal{A}))$   
 ⟨proof⟩

**lemma** *reg-funas*:  
 $\mathcal{L} \mathcal{A} \subseteq \mathcal{T}_G (\text{fset } (\text{ta-sig } (\text{ta } \mathcal{A})))$  ⟨proof⟩

**lemma** *ta-syms-lang*:  $t \in \text{ta-lang } Q \mathcal{A} \implies \text{ffunas-term } t \mid \subseteq \mid \text{ta-sig } \mathcal{A}$   
 ⟨proof⟩

**lemma** *gta-lang-Rest-states-conv*:  
 $gta\text{-}lang\ Q\ \mathcal{A} = gta\text{-}lang\ (Q\ |\cap|\ \mathcal{Q}\ \mathcal{A})\ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *reg-Rest-fin-states* [*simp*]:  
 $\mathcal{L}\ (reg\text{-}Restr\text{-}Q_f\ \mathcal{A}) = \mathcal{L}\ \mathcal{A}$   
 $\langle proof \rangle$

Deterministic tree automaton

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

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

**lemma** *ta-detE*[*elim, consumes 1*]: **assumes** *det*:  $ta\text{-}det\ \mathcal{A}$   
**shows**  $q\ |\in|\ ta\text{-}der\ \mathcal{A}\ t \Longrightarrow q'\ |\in|\ ta\text{-}der\ \mathcal{A}\ t \Longrightarrow q = q'$   $\langle proof \rangle$

**lemma** *ta-subset-states*:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow \mathcal{Q}\ \mathcal{A}\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$   
 $\langle proof \rangle$

**lemma** *ta-subset-refl*[*simp*]:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-subset-trans*:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow ta\text{-}subset\ \mathcal{B}\ \mathcal{C} \Longrightarrow ta\text{-}subset\ \mathcal{A}\ \mathcal{C}$   
 $\langle proof \rangle$

**lemma** *ta-subset-det*:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow ta\text{-}det\ \mathcal{B} \Longrightarrow ta\text{-}det\ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-der-mono'*:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow ta\text{-}der\ \mathcal{A}\ t\ |\subseteq|\ ta\text{-}der\ \mathcal{B}\ t$   
 $\langle proof \rangle$

**lemma** *ta-lang-mono'*:  $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow Q_{\mathcal{A}}\ |\subseteq|\ Q_{\mathcal{B}} \Longrightarrow ta\text{-}lang\ Q_{\mathcal{A}}\ \mathcal{A} \subseteq ta\text{-}lang\ Q_{\mathcal{B}}\ \mathcal{B}$   
 $\langle proof \rangle$

**lemma** *ta-restrict-subset*:  $ta\text{-}subset\ (ta\text{-}restrict\ \mathcal{A}\ Q)\ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-restrict-states-Q*:  $\mathcal{Q}\ (ta\text{-}restrict\ \mathcal{A}\ Q)\ |\subseteq|\ \mathcal{Q}$   
 $\langle proof \rangle$

**lemma** *ta-restrict-states*:  $\mathcal{Q}\ (ta\text{-}restrict\ \mathcal{A}\ Q)\ |\subseteq|\ \mathcal{Q}\ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-restrict-states-eq-imp-eq* [*simp*]:  
**assumes** *eq*:  $\mathcal{Q} (ta\text{-restrict } \mathcal{A} Q) = \mathcal{Q} \mathcal{A}$   
**shows**  $ta\text{-restrict } \mathcal{A} Q = \mathcal{A}$  *<proof>*

**lemma** *ta-der-ta-derict-states*:  
*fvars-term*  $t \mid \subseteq \mid Q \implies q \mid \in \mid ta\text{-der } (ta\text{-restrict } \mathcal{A} Q) t \implies q \mid \in \mid Q$   
*<proof>*

**lemma** *ta-derict-ruleI* [*intro*]:  
*TA-rule*  $f\ qs\ q \mid \in \mid rules\ \mathcal{A} \implies fset\text{-of-list } qs \mid \subseteq \mid Q \implies q \mid \in \mid Q \implies TA\text{-rule } f\ qs$   
 $q \mid \in \mid rules\ (ta\text{-restrict } \mathcal{A} Q)$   
*<proof>*

Reachable and productive states: There always is a trim automaton

**lemma** *finite-ta-reachable* [*simp*]:  
*finite*  $\{q. \exists t. ground\ t \wedge q \mid \in \mid ta\text{-der } \mathcal{A} t\}$   
*<proof>*

**lemma** *ta-reachable-states*:  
 $ta\text{-reachable } \mathcal{A} \mid \subseteq \mid \mathcal{Q} \mathcal{A}$   
*<proof>*

**lemma** *ta-reachableE*:  
**assumes**  $q \mid \in \mid ta\text{-reachable } \mathcal{A}$   
**obtains**  $t$  **where**  $ground\ t\ q \mid \in \mid ta\text{-der } \mathcal{A} t$   
*<proof>*

**lemma** *ta-reachable-gtermE* [*elim*]:  
**assumes**  $q \mid \in \mid ta\text{-reachable } \mathcal{A}$   
**obtains**  $t$  **where**  $q \mid \in \mid ta\text{-der } \mathcal{A} (term\text{-of-gterm } t)$   
*<proof>*

**lemma** *ta-reachableI* [*intro*]:  
**assumes**  $ground\ t$  **and**  $q \mid \in \mid ta\text{-der } \mathcal{A} t$   
**shows**  $q \mid \in \mid ta\text{-reachable } \mathcal{A}$   
*<proof>*

**lemma** *ta-reachable-gtermI* [*intro*]:  
 $q \mid \in \mid ta\text{-der } \mathcal{A} (term\text{-of-gterm } t) \implies q \mid \in \mid ta\text{-reachable } \mathcal{A}$   
*<proof>*

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

**lemma** *ta-reachable-rule-gtermE*:  
**assumes**  $Q \mathcal{A} \mid\subseteq\mid$  *ta-reachable*  $\mathcal{A}$   
**and** *TA-rule*  $f \text{ qs } q \mid\in\mid$  *rules*  $\mathcal{A}$   
**obtains**  $t$  **where**  $\text{groot } t = (f, \text{length } \text{qs}) \text{ } q \mid\in\mid$  *ta-der*  $\mathcal{A}$  (*term-of-gterm*  $t$ )  
 $\langle$ *proof* $\rangle$

**lemma** *ta-reachableI-eps'*:  
**assumes** *reach*:  $q \mid\in\mid$  *ta-reachable*  $\mathcal{A}$   
**and** *eps*:  $(q, q') \mid\in\mid$   $(\text{eps } \mathcal{A})^{\mid+\mid}$   
**shows**  $q' \mid\in\mid$  *ta-reachable*  $\mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-reachableI-eps*:  
**assumes** *reach*:  $q \mid\in\mid$  *ta-reachable*  $\mathcal{A}$   
**and** *eps*:  $(q, q') \mid\in\mid$  *eps*  $\mathcal{A}$   
**shows**  $q' \mid\in\mid$  *ta-reachable*  $\mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma** *finite-ta-productive*:  
*finite*  $\{p. \exists q q' C. p = q \wedge q' \mid\in\mid$  *ta-der*  $\mathcal{A} \ C \langle \text{Var } q \rangle \wedge q' \mid\in\mid P\}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-productiveE*: **assumes**  $q \mid\in\mid$  *ta-productive*  $P \mathcal{A}$   
**obtains**  $q' C$  **where**  $q' \mid\in\mid$  *ta-der*  $\mathcal{A} \ (C \langle \text{Var } q \rangle) \text{ } q' \mid\in\mid P$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-productiveI*:  
**assumes**  $q' \mid\in\mid$  *ta-der*  $\mathcal{A} \ (C \langle \text{Var } q \rangle) \text{ } q' \mid\in\mid P$   
**shows**  $q \mid\in\mid$  *ta-productive*  $P \mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-productiveI'*:  
**assumes**  $q \mid\in\mid$  *ta-der*  $\mathcal{A} \ (C \langle \text{Var } p \rangle) \text{ } q \mid\in\mid$  *ta-productive*  $P \mathcal{A}$   
**shows**  $p \mid\in\mid$  *ta-productive*  $P \mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-productive-setI*:  
 $q \mid\in\mid P \implies q \mid\in\mid$  *ta-productive*  $P \mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-reachable-empty-rules* [*simp*]:  
*rules*  $\mathcal{A} = \{\mid\}$   $\implies$  *ta-reachable*  $\mathcal{A} = \{\mid\}$   
 $\langle$ *proof* $\rangle$

**lemma** *ta-reachable-mono*:  
*ta-subset*  $\mathcal{A} \ \mathcal{B} \implies$  *ta-reachable*  $\mathcal{A} \ \mid\subseteq\mid$  *ta-reachable*  $\mathcal{B}$   $\langle$ *proof* $\rangle$

**lemma** *ta-reachable-rhs-states*:

$ta\text{-reachable } \mathcal{A} \mid \subseteq \mid ta\text{-rhs-states } \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-reachable-eps*:

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

**lemma** *ta-der-only-reach*:

**assumes**  $fvars\text{-term } t \mid \subseteq \mid ta\text{-reachable } \mathcal{A}$   
**shows**  $ta\text{-der } \mathcal{A} \ t = ta\text{-der } (ta\text{-only-reach } \mathcal{A}) \ t$  (**is**  $?LS = ?RS$ )  
 $\langle proof \rangle$

**lemma** *ta-der-gterm-only-reach*:

$ta\text{-der } \mathcal{A} \ (term\text{-of-gterm } t) = ta\text{-der } (ta\text{-only-reach } \mathcal{A}) \ (term\text{-of-gterm } t)$   
 $\langle proof \rangle$

**lemma** *ta-reachable-ta-only-reach [simp]*:

$ta\text{-reachable } (ta\text{-only-reach } \mathcal{A}) = ta\text{-reachable } \mathcal{A}$  (**is**  $?LS = ?RS$ )  
 $\langle proof \rangle$

**lemma** *ta-only-reach-reachable*:

$Q \ (ta\text{-only-reach } \mathcal{A}) \mid \subseteq \mid ta\text{-reachable } (ta\text{-only-reach } \mathcal{A})$   
 $\langle proof \rangle$

**lemma** *gta-only-reach-lang*:

$gta\text{-lang } Q \ (ta\text{-only-reach } \mathcal{A}) = gta\text{-lang } Q \ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *L-only-reach*:  $\mathcal{L} \ (reg\text{-reach } R) = \mathcal{L} \ R$

$\langle proof \rangle$

**lemma** *ta-only-reach-lang*:

$ta\text{-lang } Q \ (ta\text{-only-reach } \mathcal{A}) = ta\text{-lang } Q \ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-prod-epsD*:

$(p, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \implies q \mid \in \mid ta\text{-productive } P \ \mathcal{A} \implies p \mid \in \mid ta\text{-productive } P \ \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *ta-only-prod-eps*:

$(p, q) \mid \in \mid (eps \mathcal{A}) \mid^+ \implies q \mid \in \mid ta\text{-productive } P \ \mathcal{A} \implies (p, q) \mid \in \mid (eps \ (ta\text{-only-prod } P \ \mathcal{A})) \mid^+$

*<proof>*

**lemma** *ta-der-only-prod*:

$q \in | \text{ta-der } \mathcal{A} \ t \implies q \in | \text{ta-productive } P \ \mathcal{A} \implies q \in | \text{ta-der } (\text{ta-only-prod } P \ \mathcal{A}) \ t$   
*<proof>*

**lemma** *ta-der-ta-only-prod-ta-der*:

$q \in | \text{ta-der } (\text{ta-only-prod } P \ \mathcal{A}) \ t \implies q \in | \text{ta-der } \mathcal{A} \ t$   
*<proof>*

**lemma** *gta-only-prod-lang*:

$\text{gta-lang } Q \ (\text{ta-only-prod } Q \ \mathcal{A}) = \text{gta-lang } Q \ \mathcal{A} \ (\text{is } \text{gta-lang } Q \ ?\mathcal{A} = -)$   
*<proof>*

**lemma** *L-only-prod*:  $\mathcal{L} \ (\text{reg-prod } R) = \mathcal{L} \ R$

*<proof>*

**lemma** *ta-only-prod-lang*:

$\text{ta-lang } Q \ (\text{ta-only-prod } Q \ \mathcal{A}) = \text{ta-lang } Q \ \mathcal{A}$   
*<proof>*

**lemma** *ta-productive-ta-only-prod [simp]*:

$\text{ta-productive } P \ (\text{ta-only-prod } P \ \mathcal{A}) = \text{ta-productive } P \ \mathcal{A} \ (\text{is } ?LS = ?RS)$   
*<proof>*

**lemma** *ta-only-prod-productive*:

$Q \ (\text{ta-only-prod } P \ \mathcal{A}) \subseteq | \text{ta-productive } P \ (\text{ta-only-prod } P \ \mathcal{A})$   
*<proof>*

**lemma** *ta-only-prod-reachable*:

**assumes** *all-reach*:  $Q \ \mathcal{A} \subseteq | \text{ta-reachable } \mathcal{A}$

**shows**  $Q \ (\text{ta-only-prod } P \ \mathcal{A}) \subseteq | \text{ta-reachable } (\text{ta-only-prod } P \ \mathcal{A}) \ (\text{is } ?Ls \subseteq | ?Rs)$   
*<proof>*

**lemma** *ta-prod-reach-subset*:

$\text{ta-subset } (\text{ta-only-prod } P \ (\text{ta-only-reach } \mathcal{A})) \ \mathcal{A}$   
*<proof>*

**lemma** *ta-prod-reach-states*:

$Q \ (\text{ta-only-prod } P \ (\text{ta-only-reach } \mathcal{A})) \subseteq | Q \ \mathcal{A}$   
*<proof>*

**lemma** *ta-productive-aux*:

**assumes**  $\mathcal{Q} \mathcal{A} \mid \subseteq \mid$  *ta-reachable*  $\mathcal{A} q \mid \in \mid$  *ta-der*  $\mathcal{A} (C \langle t \rangle)$   
**shows**  $\exists C'. \text{ground-ctxt } C' \wedge q \mid \in \mid$  *ta-der*  $\mathcal{A} (C' \langle t \rangle)$  *<proof>*

**lemma** *ta-productive-def'*:

**assumes**  $\mathcal{Q} \mathcal{A} \mid \subseteq \mid$  *ta-reachable*  $\mathcal{A}$   
**shows** *ta-productive*  $\mathcal{Q} \mathcal{A} = \{ \mid q \mid q q' C. \text{ground-ctxt } C \wedge q' \mid \in \mid$  *ta-der*  $\mathcal{A} (C \langle \text{Var } q \rangle) \wedge q' \mid \in \mid \mathcal{Q} \mid \}$   
*<proof>*

**lemma** *trim-gta-lang*: *gta-lang*  $\mathcal{Q} (\text{trim-ta } \mathcal{Q} \mathcal{A}) = \text{gta-lang } \mathcal{Q} \mathcal{A}$   
*<proof>*

**lemma** *trim-ta-subset*: *ta-subset*  $(\text{trim-ta } \mathcal{Q} \mathcal{A}) \mathcal{A}$   
*<proof>*

**theorem** *trim-ta*: *ta-is-trim*  $\mathcal{Q} (\text{trim-ta } \mathcal{Q} \mathcal{A})$  *<proof>*

**lemma** *reg-is-trim-trim-reg* [*simp*]: *reg-is-trim*  $(\text{trim-reg } R)$   
*<proof>*

**lemma** *trim-reg-reach* [*simp*]:  
 $\mathcal{Q}_r (\text{trim-reg } A) \mid \subseteq \mid$  *ta-reachable*  $(\text{ta } (\text{trim-reg } A))$   
*<proof>*

**lemma** *trim-reg-prod* [*simp*]:  
 $\mathcal{Q}_r (\text{trim-reg } A) \mid \subseteq \mid$  *ta-productive*  $(\text{fin } (\text{trim-reg } A)) (\text{ta } (\text{trim-reg } A))$   
*<proof>*

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

**lemma** *L-trim-ta-sig*:

**assumes** *reg-is-trim*  $R \mathcal{L} R \subseteq \mathcal{T}_G (\text{fset } \mathcal{F})$   
**shows** *ta-sig*  $(\text{ta } R) \mid \subseteq \mid \mathcal{F}$   
*<proof>*

Map function over TA rules which change states/signature

**lemma** *map-ta-rule-iff*:

*map-ta-rule*  $f g \mid \uparrow \mid \Delta = \{ \mid \text{TA-rule } (g h) (\text{map } f qs) (f q) \mid h qs q. \text{TA-rule } h qs q \mid \in \mid \Delta \mid \}$   
*<proof>*

**lemma** *L-trim*:  $\mathcal{L} (\text{trim-reg } R) = \mathcal{L} R$   
*<proof>*

**lemma** *fmap-funs-ta-def'*:

*fmap-funs-ta*  $h \mathcal{A} = TA \{|(h f) qs \rightarrow q \mid f qs q. f qs \rightarrow q \mid \in \mid \text{rules } \mathcal{A} \mid\} (eps \mathcal{A})$   
*<proof>*

**lemma** *fmap-states-ta-def'*:

*fmap-states-ta*  $h \mathcal{A} = TA \{|f (map h qs) \rightarrow h q \mid f qs q. f qs \rightarrow q \mid \in \mid \text{rules } \mathcal{A} \mid\}$   
*(map-both h |' eps A)*  
*<proof>*

**lemma** *fmap-states [simp]*:

$\mathcal{Q} (fmap-states-ta h \mathcal{A}) = h \mid' \mathcal{Q} \mathcal{A}$   
*<proof>*

**lemma** *fmap-states-ta-sig [simp]*:

*ta-sig (fmap-states-ta f A) = ta-sig A*  
*<proof>*

**lemma** *fmap-states-ta-eps-wit*:

**assumes**  $(h p, q) \mid \in \mid (map-both h \mid' eps \mathcal{A}) \mid^+ \mid \text{finj-on } h (\mathcal{Q} \mathcal{A}) p \mid \in \mid \mathcal{Q} \mathcal{A}$   
**obtains**  $q'$  **where**  $q = h q' (p, q') \mid \in \mid (eps \mathcal{A}) \mid^+ \mid q' \mid \in \mid \mathcal{Q} \mathcal{A}$   
*<proof>*

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

**assumes**  $\mathcal{Q} \mathcal{A} \mid \subseteq \mid \mathcal{B} \mid \text{finj-on } h \mathcal{B}$   
**and**  $q \mid \in \mid \text{ta-der } (fmap-states-ta h \mathcal{A}) (term-of-gterm t)$   
**shows**  $the-finw-into \mathcal{B} h q \mid \in \mid \text{ta-der } \mathcal{A} (term-of-gterm t) \mid \langle proof \rangle$

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

**assumes**  $\text{finj-on } h (\mathcal{Q} \mathcal{A}) q \mid \in \mid \text{ta-der } (fmap-states-ta h \mathcal{A}) (term-of-gterm t)$   
**shows**  $the-finw-into (\mathcal{Q} \mathcal{A}) h q \mid \in \mid \text{ta-der } \mathcal{A} (term-of-gterm t)$   
*<proof>*

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

**assumes**  $q \mid \in \mid \text{ta-der } \mathcal{A} (term-of-gterm t)$   
**shows**  $h q \mid \in \mid \text{ta-der } (fmap-states-ta h \mathcal{A}) (term-of-gterm t) \mid \langle proof \rangle$

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

**assumes**  $\text{finj-on } h (\mathcal{Q} \mathcal{A})$   
**shows**  $\text{ta-der } (fmap-states-ta h \mathcal{A}) (term-of-gterm t) = h \mid' \mid \text{ta-der } \mathcal{A} (term-of-gterm t)$   
*<proof>*

**lemma** *fmap-states-ta-det*:

**assumes**  $\text{finj-on } f (\mathcal{Q} \mathcal{A})$   
**shows**  $\text{ta-det } (fmap-states-ta f \mathcal{A}) = \text{ta-det } \mathcal{A} (\text{is } ?Ls = ?Rs)$   
*<proof>*

**lemma** *fmap-states-ta-lang*:

$\text{finj-on } f \ (\mathcal{Q} \ \mathcal{A}) \implies \mathcal{Q} \ |\subseteq| \ \mathcal{Q} \ \mathcal{A} \implies \text{gta-lang } (f \ |^{\dagger} \ \mathcal{Q}) \ (\text{fmap-states-ta } f \ \mathcal{A}) = \text{gta-lang } \mathcal{Q} \ \mathcal{A}$   
 ⟨proof⟩

**lemma** *fmap-states-ta-lang2*:

$\text{finj-on } f \ (\mathcal{Q} \ \mathcal{A} \ |\cup| \ \mathcal{Q}) \implies \text{gta-lang } (f \ |^{\dagger} \ \mathcal{Q}) \ (\text{fmap-states-ta } f \ \mathcal{A}) = \text{gta-lang } \mathcal{Q} \ \mathcal{A}$   
 ⟨proof⟩

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

$\text{funs-ta } \mathcal{A} = \{|f \ |f \ \text{qs } q. \ \text{TA-rule } f \ \text{qs } q \ | \in| \ \text{rules } \mathcal{A}|\}$

**lemma** *funs-ta[code]*:

$\text{funs-ta } \mathcal{A} = (\lambda r. \ \text{case } r \ \text{of } \ \text{TA-rule } f \ \text{ps } p \Rightarrow f) \ |^{\dagger} \ (\text{rules } \mathcal{A}) \ (\text{is } ?Ls = ?Rs)$   
 ⟨proof⟩

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

$\text{finite } \{|f. \ \exists \ \text{qs } q. \ \text{TA-rule } f \ \text{qs } q \ | \in| \ \text{rules } \mathcal{A}|\}$   
 ⟨proof⟩

**lemma** *funs-taE [elim]*:

**assumes**  $f \ | \in| \ \text{funs-ta } \mathcal{A}$   
**obtains**  $\text{ps } p$  **where**  $\text{TA-rule } f \ \text{ps } p \ | \in| \ \text{rules } \mathcal{A}$  ⟨proof⟩

**lemma** *funs-taI [intro]*:

$\text{TA-rule } f \ \text{ps } p \ | \in| \ \text{rules } \mathcal{A} \implies f \ | \in| \ \text{funs-ta } \mathcal{A}$   
 ⟨proof⟩

**lemma** *fmap-funs-ta-cong*:

$(\bigwedge x. \ x \ | \in| \ \text{funs-ta } \mathcal{A} \implies h \ x = k \ x) \implies \mathcal{A} = \mathcal{B} \implies \text{fmap-funs-ta } h \ \mathcal{A} = \text{fmap-funs-ta } k \ \mathcal{B}$   
 ⟨proof⟩

**lemma** *[simp]*:  $\{| \ \text{TA-rule } f \ \text{qs } q \ | \ \text{qs } q. \ \text{TA-rule } f \ \text{qs } q \ | \in| \ X |\} = X$

⟨proof⟩

**lemma** *fmap-funs-ta-id [simp]*:

$\text{fmap-funs-ta } \text{id } \mathcal{A} = \mathcal{A}$  ⟨proof⟩

**lemma** *fmap-states-ta-id [simp]*:

$\text{fmap-states-ta } \text{id } \mathcal{A} = \mathcal{A}$   
 ⟨proof⟩

**lemmas**  $\text{fmap-funs-ta-id}' \ [\text{simp}] = \text{fmap-funs-ta-id}[\text{unfolded id-def}]$

**lemma** *fmap-funs-ta-comp*:

$\text{fmap-funs-ta } h \ (\text{fmap-funs-ta } k \ \mathcal{A}) = \text{fmap-funs-ta } (h \circ k) \ \mathcal{A}$   
 ⟨proof⟩

**lemma** *fmap-funs-reg-comp*:

$$\text{fmap-funs-reg } h \text{ (fmap-funs-reg } k \text{ } A) = \text{fmap-funs-reg } (h \circ k) \text{ } A$$

*<proof>*

**lemma** *fmap-states-ta-comp*:

$$\text{fmap-states-ta } h \text{ (fmap-states-ta } k \text{ } A) = \text{fmap-states-ta } (h \circ k) \text{ } A$$

*<proof>*

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

$$\text{funs-ta (fmap-funs-ta } f \text{ } A) = f \text{ |}^\dagger \text{ funs-ta } A$$

*<proof>*

**lemma** *ta-der-funs-ta*:

$$q \text{ |}\in\text{| ta-der } A \text{ } t \implies \text{ffuns-term } t \text{ |}\subseteq\text{| funs-ta } A$$

*<proof>*

**lemma** *ta-der-fmap-funs-ta*:

$$q \text{ |}\in\text{| ta-der } A \text{ } t \implies q \text{ |}\in\text{| ta-der (fmap-funs-ta } f \text{ } A) \text{ (map-funs-term } f \text{ } t)$$

*<proof>*

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

**assumes**  $q \text{ |}\in\text{| ta-der } A \text{ } t$

**shows**  $h \text{ } q \text{ |}\in\text{| ta-der (fmap-states-ta } h \text{ } A) \text{ (map-vars-term } h \text{ } t)$

*<proof>*

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

**shows**  $f \text{ |}^\dagger \text{ ta-der } A \text{ (term-of-gterm } s) \text{ |}\subseteq\text{| ta-der (fmap-states-ta } f \text{ } A) \text{ (term-of-gterm } s)$

*<proof>*

**lemma** *ta-der-fmap-states-ta-mono2*:

**assumes** *finj-on*  $f \text{ (} Q \text{ } A)$

**shows**  $\text{ta-der (fmap-states-ta } f \text{ } A) \text{ (term-of-gterm } s) \text{ |}\subseteq\text{| } f \text{ |}^\dagger \text{ ta-der } A \text{ (term-of-gterm } s)$

*<proof>*

**lemma** *fmap-funs-ta-der'*:

$q \text{ |}\in\text{| ta-der (fmap-funs-ta } h \text{ } A) \text{ } t \implies \exists t'. q \text{ |}\in\text{| ta-der } A \text{ } t' \wedge \text{map-funs-term } h \text{ } t' = t$

*<proof>*

**lemma** *fmap-funs-gta-lang*:

$$\text{gta-lang } Q \text{ (fmap-funs-ta } h \text{ } \mathcal{A}) = \text{map-gterm } h \text{ ' gta-lang } Q \text{ } \mathcal{A} \text{ (is ?Ls = ?Rs)}$$

*<proof>*

**lemma** *fmap-funs- $\mathcal{L}$* :

$$\mathcal{L} \text{ (fmap-funs-reg } h \text{ } R) = \text{map-gterm } h \text{ ' } \mathcal{L} \text{ } R$$

*<proof>*

**lemma** *ta-states-fmap-funs-ta* [simp]:  $\mathcal{Q} (\text{fmap-funs-ta } f A) = \mathcal{Q} A$   
 ⟨proof⟩

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

**lemma** *fin-in-states*:  
 $\text{fin} (\text{reg-Restr-}Q_f R) \mid\subseteq\mid \mathcal{Q}_r (\text{reg-Restr-}Q_f R)$   
 ⟨proof⟩

**lemma** *fmap-states-reg-Restr- $Q_f$ -fin*:  
 $\text{finj-on } f (\mathcal{Q} A) \implies \text{fin} (\text{fmap-states-reg } f (\text{reg-Restr-}Q_f R)) \mid\subseteq\mid \mathcal{Q}_r (\text{fmap-states-reg } f (\text{reg-Restr-}Q_f R))$   
 ⟨proof⟩

**lemma**  *$\mathcal{L}$ -fmap-states-reg-Inl-Inr* [simp]:  
 $\mathcal{L} (\text{fmap-states-reg Inl } R) = \mathcal{L} R$   
 $\mathcal{L} (\text{fmap-states-reg Inr } R) = \mathcal{L} R$   
 ⟨proof⟩

**lemma** *finite-Collect-prod-ta-rules*:  
 $\text{finite} \{f \text{ qs} \rightarrow (a, b) \mid f \text{ qs } a \ b. \ f \text{ map fst qs} \rightarrow a \mid\in\mid \text{rules } \mathcal{A} \wedge f \text{ map snd qs} \rightarrow b \mid\in\mid \text{rules } \mathfrak{B}\} \text{ (is finite ?set)}$   
 ⟨proof⟩

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

**lemma** *finite-prod-epsLp*:  
 $\text{finite} (\text{Collect} (\text{prod-epsLp } \mathcal{A} \ \mathcal{B}))$   
 ⟨proof⟩

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

**lemmas** *finite-prod-eps* [simp] = *finite-prod-epsLp[unfolded prod-epsLp-def] finite-prod-epsRp[unfolded prod-epsRp-def]*

**lemma** [simp]:  $f \text{ qs} \rightarrow q \mid\in\mid \text{rules} (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \iff f \text{ qs} \rightarrow q \mid\in\mid \text{prod-ta-rules } \mathcal{A} \ \mathcal{B}$   
 $r \mid\in\mid \text{rules} (\text{prod-ta } \mathcal{A} \ \mathcal{B}) \iff r \mid\in\mid \text{prod-ta-rules } \mathcal{A} \ \mathcal{B}$   
 ⟨proof⟩

**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⟩

**lemma** *prod-ta-det*:  
 assumes *ta-det*  $\mathcal{A}$  and *ta-det*  $\mathcal{B}$

**shows**  $ta\text{-det}$  ( $prod\text{-ta}$   $\mathcal{A}$   $\mathcal{B}$ )  
 $\langle proof \rangle$

**lemma**  $prod\text{-ta}\text{-sig}$ :  
 $ta\text{-sig}$  ( $prod\text{-ta}$   $\mathcal{A}$   $\mathcal{B}$ )  $\subseteq$   $ta\text{-sig}$   $\mathcal{A}$   $\cup$   $ta\text{-sig}$   $\mathcal{B}$   
 $\langle proof \rangle$

**lemma**  $from\text{-prod}\text{-eps}$ :  
 $(p, q) \in | (eps (prod\text{-ta} \mathcal{A} \mathcal{B})) |^+ \implies (snd\ p, snd\ q) \notin | (eps \mathcal{B}) |^+ \implies snd\ p =$   
 $snd\ q \wedge (fst\ p, fst\ q) \in | (eps \mathcal{A}) |^+$   
 $(p, q) \in | (eps (prod\text{-ta} \mathcal{A} \mathcal{B})) |^+ \implies (fst\ p, fst\ q) \notin | (eps \mathcal{A}) |^+ \implies fst\ p = fst$   
 $q \wedge (snd\ p, snd\ q) \in | (eps \mathcal{B}) |^+$   
 $\langle proof \rangle$

**lemma**  $to\text{-prod}\text{-eps}\mathcal{A}$ :  
 $(p, q) \in | (eps \mathcal{A}) |^+ \implies r \in \mathcal{Q} \mathcal{B} \implies ((p, r), (q, r)) \in | (eps (prod\text{-ta} \mathcal{A} \mathcal{B})) |^+$   
 $\langle proof \rangle$

**lemma**  $to\text{-prod}\text{-eps}\mathcal{B}$ :  
 $(p, q) \in | (eps \mathcal{B}) |^+ \implies r \in \mathcal{Q} \mathcal{A} \implies ((r, p), (r, q)) \in | (eps (prod\text{-ta} \mathcal{A} \mathcal{B})) |^+$   
 $\langle proof \rangle$

**lemma**  $to\text{-prod}\text{-eps}$ :  
 $(p, q) \in | (eps \mathcal{A}) |^+ \implies (p', q') \in | (eps \mathcal{B}) |^+ \implies ((p, p'), (q, q')) \in | (eps$   
 $(prod\text{-ta} \mathcal{A} \mathcal{B})) |^+$   
 $\langle proof \rangle$

**lemma**  $prod\text{-ta}\text{-der}\text{-to}\text{-}\mathcal{A}\text{-}\mathcal{B}\text{-}\text{der}1$ :  
**assumes**  $q \in | ta\text{-der} (prod\text{-ta} \mathcal{A} \mathcal{B}) (term\text{-of}\text{-gterm } t)$   
**shows**  $fst\ q \in | ta\text{-der} \mathcal{A} (term\text{-of}\text{-gterm } t) \langle proof \rangle$

**lemma**  $prod\text{-ta}\text{-der}\text{-to}\text{-}\mathcal{A}\text{-}\mathcal{B}\text{-}\text{der}2$ :  
**assumes**  $q \in | ta\text{-der} (prod\text{-ta} \mathcal{A} \mathcal{B}) (term\text{-of}\text{-gterm } t)$   
**shows**  $snd\ q \in | ta\text{-der} \mathcal{B} (term\text{-of}\text{-gterm } t) \langle proof \rangle$

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

**lemma**  $prod\text{-ta}\text{-der}$ :  
 $q \in | ta\text{-der} (prod\text{-ta} \mathcal{A} \mathcal{B}) (term\text{-of}\text{-gterm } t) \iff$   
 $fst\ q \in | ta\text{-der} \mathcal{A} (term\text{-of}\text{-gterm } t) \wedge snd\ q \in | ta\text{-der} \mathcal{B} (term\text{-of}\text{-gterm } t)$   
 $\langle proof \rangle$

**lemma**  $intersect\text{-ta}\text{-gta}\text{-lang}$ :  
 $gta\text{-lang} (Q_{\mathcal{A}} \times Q_{\mathcal{B}}) (prod\text{-ta} \mathcal{A} \mathcal{B}) = gta\text{-lang } Q_{\mathcal{A}} \mathcal{A} \cap gta\text{-lang } Q_{\mathcal{B}} \mathcal{B}$   
 $\langle proof \rangle$

**lemma**  $\mathcal{L}$ -intersect:  $\mathcal{L} (\text{reg-intersect } R L) = \mathcal{L} R \cap \mathcal{L} L$   
 ⟨proof⟩

**lemma** intersect-ta-ta-lang:  
 $\text{ta-lang } (Q_A \times | Q_B) (\text{prod-ta } \mathcal{A} \mathcal{B}) = \text{ta-lang } Q_A \mathcal{A} \cap \text{ta-lang } Q_B \mathcal{B}$   
 ⟨proof⟩

**lemma** ta-union-ta-subset:  
 $\text{ta-subset } \mathcal{A} (\text{ta-union } \mathcal{A} \mathcal{B}) \text{ ta-subset } \mathcal{B} (\text{ta-union } \mathcal{A} \mathcal{B})$   
 ⟨proof⟩

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

**lemma** ta-union-sig [simp]:  
 $\text{ta-sig } (\text{ta-union } \mathcal{A} \mathcal{B}) = \text{ta-sig } \mathcal{A} \cup | \text{ta-sig } \mathcal{B}$   
 ⟨proof⟩

**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  $(p, q) \in | (\text{eps } \mathcal{A})|^{+} \vee (p, q) \in | (\text{eps } \mathcal{B})|^{+}$  ⟨proof⟩

**lemma** eps-ta-union-eps [simp]:  
 $(p, q) \in | (\text{eps } \mathcal{A})|^{+} \implies (p, q) \in | (\text{eps } (\text{ta-union } \mathcal{A} \mathcal{B}))|^{+}$   
 $(p, q) \in | (\text{eps } \mathcal{B})|^{+} \implies (p, q) \in | (\text{eps } (\text{ta-union } \mathcal{A} \mathcal{B}))|^{+}$   
 ⟨proof⟩

**lemma** disj-states-eps [simp]:  
 $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\} \implies f \text{ ps } \rightarrow p \in | \text{rules } \mathcal{A} \implies (p, q) \in | (\text{eps } \mathcal{B})|^{+} \longleftrightarrow \text{False}$   
 $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\} \implies f \text{ ps } \rightarrow p \in | \text{rules } \mathcal{B} \implies (p, q) \in | (\text{eps } \mathcal{A})|^{+} \longleftrightarrow \text{False}$   
 ⟨proof⟩

**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  $q \in | \text{ta-der } \mathcal{A} t \vee q \in | \text{ta-der } \mathcal{B} t$  ⟨proof⟩

**lemma** ta-union-der-disj-states':  
 assumes  $\mathcal{Q} \mathcal{A} \cap | \mathcal{Q} \mathcal{B} = \{||\}$   
 shows  $\text{ta-der } (\text{ta-union } \mathcal{A} \mathcal{B}) t = \text{ta-der } \mathcal{A} t \cup | \text{ta-der } \mathcal{B} t$   
 ⟨proof⟩

**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  $\text{gta-lang } (Q_A \cup | Q_B) (\text{ta-union } \mathcal{A} \mathcal{B}) = \text{gta-lang } Q_A \mathcal{A} \cup \text{gta-lang } Q_B \mathcal{B}$   
 (is ?Ls = ?Rs)  
 ⟨proof⟩

**lemma**  $\mathcal{L}$ -union:  $\mathcal{L} (\text{reg-union } R L) = \mathcal{L} R \cup \mathcal{L} L$   
 ⟨proof⟩

**lemma** *reg-union-states*:  
 $\mathcal{Q}_r (\text{reg-union } A B) = (\text{Inl } |^q \mathcal{Q}_r A) \cup (\text{Inr } |^q \mathcal{Q}_r B)$   
 ⟨proof⟩

**lemma** *ta-empty* [simp]:  
 $\text{ta-empty } Q \mathcal{A} = (\text{gta-lang } Q \mathcal{A} = \{\})$   
 ⟨proof⟩

**lemma** *reg-empty* [simp]:  
 $\text{reg-empty } R = (\mathcal{L} R = \{\})$   
 ⟨proof⟩

Epsilon free automaton

**lemma** *finite-eps-free-rulep* [simp]:  
 $\text{finite } (\text{Collect } (\text{eps-free-rulep } \mathcal{A}))$   
 ⟨proof⟩

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

**lemma** *ta-res-eps-free*:  
 $\text{ta-der } (\text{eps-free } \mathcal{A}) (\text{term-of-gterm } t) = \text{ta-der } \mathcal{A} (\text{term-of-gterm } t) (\text{is ?Ls} = \text{?Rs})$   
 ⟨proof⟩

**lemma** *ta-lang-eps-free* [simp]:  
 $\text{gta-lang } Q (\text{eps-free } \mathcal{A}) = \text{gta-lang } Q \mathcal{A}$   
 ⟨proof⟩

**lemma**  $\mathcal{L}$ -eps-free:  $\mathcal{L} (\text{eps-free-reg } R) = \mathcal{L} R$   
 ⟨proof⟩

Sufficient criterion for containment

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

**lemma** *ta-contains-aux-state-set*:  
**assumes** *ta-contains-aux*  $\mathcal{F} Q \mathcal{A} Q t \in \mathcal{T}_G \mathcal{F}$   
**shows**  $\exists q. q \ |\in| Q \wedge q \ |\in| \text{ta-der } \mathcal{A} (\text{term-of-gterm } t)$  ⟨proof⟩

**lemma** *ta-contains-aux-mono*:

**assumes** *ta-subset*  $\mathcal{A} \mathcal{B}$  **and**  $Q_2 \mid\subseteq\mid Q_2'$

**shows** *ta-contains-aux*  $\mathcal{F} Q_1 \mathcal{A} Q_2 \implies \text{ta-contains-aux } \mathcal{F} Q_1 \mathcal{B} Q_2'$

*<proof>*

**definition** *ta-contains* ::  $(f \times \text{nat}) \text{ set} \Rightarrow (f \times \text{nat}) \text{ set} \Rightarrow ('q, 'f) \text{ ta} \Rightarrow 'q \text{ fset} \Rightarrow 'q \text{ fset} \Rightarrow \text{bool}$

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

**lemma** *ta-contains-mono*:

**assumes** *ta-subset*  $\mathcal{A} \mathcal{B}$  **and**  $Q_f \mid\subseteq\mid Q_f'$

**shows** *ta-contains*  $\mathcal{F} \mathcal{G} \mathcal{A} Q Q_f \implies \text{ta-contains } \mathcal{F} \mathcal{G} \mathcal{B} Q Q_f'$

*<proof>*

**lemma** *ta-contains-both*:

**assumes** *contain*: *ta-contains*  $\mathcal{F} \mathcal{G} \mathcal{A} Q Q_f$

**shows**  $\bigwedge t. \text{groot } t \in \mathcal{G} \implies \bigcup (\text{funas-gterm } ' \text{ set } (\text{gargs } t)) \subseteq \mathcal{F} \implies t \in \text{gta-lang } Q_f \mathcal{A}$

*<proof>*

**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>*

Relabeling, map finite set to natural numbers

**lemma** *map-fset-to-nat-inj*:

**assumes**  $Y \mid\subseteq\mid X$

**shows** *finj-on* (*map-fset-to-nat*  $X$ )  $Y$

*<proof>*

**lemma** *map-fset-fset-to-nat-inj*:

**assumes**  $Y \mid\subseteq\mid X$

**shows** *finj-on* (*map-fset-fset-to-nat*  $X$ )  $Y$  *<proof>*

**lemma** *relabel-gta-lang* [*simp*]:

*gta-lang* (*relabel-Q<sub>f</sub>*  $Q \mathcal{A}$ ) (*relabel-ta*  $\mathcal{A}$ ) = *gta-lang*  $Q \mathcal{A}$   
*<proof>*

**lemma** *L-relabel* [*simp*]:  $\mathcal{L}$  (*relabel-reg*  $R$ ) =  $\mathcal{L} R$

*<proof>*

**lemma** *relabel-ta-lang* [*simp*]:

*ta-lang* (*relabel-Q<sub>f</sub>*  $Q \mathcal{A}$ ) (*relabel-ta*  $\mathcal{A}$ ) = *ta-lang*  $Q \mathcal{A}$

*<proof>*

**lemma** *relabel-fset-gta-lang* [simp]:

$$gta\text{-}lang\ (relabel\text{-}fset\text{-}Q_f\ Q\ \mathcal{A})\ (relabel\text{-}fset\text{-}ta\ \mathcal{A}) = gta\text{-}lang\ Q\ \mathcal{A}$$

⟨proof⟩

**lemma** *L-reliable-fset* [simp]:  $\mathcal{L}\ (reliable\text{-}fset\text{-}reg\ R) = \mathcal{L}\ R$

⟨proof⟩

**lemma** *ta-states-trim-ta*:

$$\mathcal{Q}\ (trim\text{-}ta\ Q\ \mathcal{A})\ |\subseteq|\ \mathcal{Q}\ \mathcal{A}$$

⟨proof⟩

**lemma** *trim-ta-reach*:  $\mathcal{Q}\ (trim\text{-}ta\ Q\ \mathcal{A})\ |\subseteq|\ ta\text{-}reachable\ (trim\text{-}ta\ Q\ \mathcal{A})$

⟨proof⟩

**lemma** *trim-ta-prod*:  $\mathcal{Q}\ (trim\text{-}ta\ Q\ \mathcal{A})\ |\subseteq|\ ta\text{-}productive\ Q\ (trim\text{-}ta\ Q\ \mathcal{A})$

⟨proof⟩

**lemma** *empty-gta-lang*:

$$gta\text{-}lang\ Q\ (TA\ \{\|\}\ \{\|\}) = \{\}$$

⟨proof⟩

**abbreviation** *empty-reg* where

$$empty\text{-}reg \equiv Reg\ \{\|\}\ (TA\ \{\|\}\ \{\|\})$$

**lemma** *L-empty*:

$$\mathcal{L}\ empty\text{-}reg = \{\}$$

⟨proof⟩

**lemma** *const-ta-lang*:

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

⟨proof⟩

**lemma** *run-argsD*:

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

⟨proof⟩

**lemma** *run-root-rule*:

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

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

⟨proof⟩

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

$\langle proof \rangle$

**lemma** *run-gsubt-cl*:

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

**shows**  $run \mathcal{A} (gsubt-at s p) (gsubt-at t p)$   $\langle proof \rangle$

**lemma** *run-replace-at*:

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

**and**  $ex-comp-state (gsubt-at s p) = ex-comp-state u$

**shows**  $run \mathcal{A} s[p \leftarrow u]_G t[p \leftarrow v]_G$   $\langle proof \rangle$

relating runs to derivation definition

**lemma** *run-to-comp-st-gta-der*:

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

$\langle proof \rangle$

**lemma** *run-to-rule-st-gta-der*:

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

$\langle proof \rangle$

**lemma** *run-to-gta-der-gsubt-at*:

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

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

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

$\langle proof \rangle$

**lemma** *gta-der-to-run*:

$q \mid \in \mid gta-der \mathcal{A} t \implies (\exists p qs. run \mathcal{A} (GFun (p, q) qs) t)$   $\langle proof \rangle$

**lemma** *run-ta-der-ctxt-split1*:

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

**shows**  $ex-comp-state s \mid \in \mid ta-der \mathcal{A} (ctxt-of-pos-term p (term-of-gterm t)) \langle Var$   
 $(ex-comp-state (gsubt-at s p)) \rangle$

$\langle proof \rangle$

**lemma** *run-ta-der-ctxt-split2*:

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

**shows**  $ex-comp-state s \mid \in \mid ta-der \mathcal{A} (ctxt-of-pos-term p (term-of-gterm t)) \langle Var$   
 $(ex-rule-state (gsubt-at s p)) \rangle$

$\langle proof \rangle$

**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*  $\mathcal{A} f ps = (\lambda q'. \exists qs q. TA\text{-rule } f qs q \mid \in \mid \text{rules } \mathcal{A} \wedge list\text{-all2} (|\in|) qs ps \wedge (q = q' \vee (q, q') \mid \in \mid (eps \mathcal{A})^+))$

**lemma** *ps-reachable-statespE*:

**assumes** *ps-reachable-statesp*  $\mathcal{A} f qs q$

**obtains** *ps p* **where** *TA-rule*  $f ps p \mid \in \mid \text{rules } \mathcal{A} list\text{-all2} (|\in|) ps qs (p = q \vee (p, q) \mid \in \mid (eps \mathcal{A})^+)$   
 $\langle proof \rangle$

**lemma** *ps-reachable-statesp-Q*:

*ps-reachable-statesp*  $\mathcal{A} f ps q \implies q \mid \in \mid \mathcal{Q} \mathcal{A}$   
 $\langle proof \rangle$

**lemma** *finite-Collect-ps-statep* [*simp*]:

*finite* (*Collect* (*ps-reachable-statesp*  $\mathcal{A} f ps$ )) (**is** *finite* ?*S*)  
 $\langle proof \rangle$

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

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

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

**lemma** *ps-reachable-states-fmember*:

$q' \mid \in \mid ps\text{-reachable-states } \mathcal{A} f ps \iff (\exists qs q. TA\text{-rule } f qs q \mid \in \mid \text{rules } \mathcal{A} \wedge list\text{-all2} (|\in|) qs ps \wedge (q = q' \vee (q, q') \mid \in \mid (eps \mathcal{A})^+))$   
 $\langle proof \rangle$

**lemma** *ps-reachable-statesI*:

**assumes** *TA-rule*  $f ps p \mid \in \mid \text{rules } \mathcal{A} list\text{-all2} (|\in|) ps qs (p = q \vee (p, q) \mid \in \mid (eps \mathcal{A})^+)$

**shows**  $p \mid \in \mid ps\text{-reachable-states } \mathcal{A} f qs$   
 $\langle proof \rangle$

**lemma** *ps-reachable-states-sig*:

*ps-reachable-states*  $\mathcal{A} f ps \neq \{\}\implies (f, length ps) \mid \in \mid ta\text{-sig } \mathcal{A}$   
 $\langle proof \rangle$

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*  $\implies 'a$  *FSet-Lex-Wrapper* *fset*  $\implies 'a$

*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*))  $\subseteq$   $Q_{next}$   
*<proof>*

**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*  $|\cdot|^{\dagger}$  *fPow* ( $\mathcal{Q}$   $\mathcal{A}$ ))  
*<proof>*

**context**

**includes** *fset.lifting*

**begin**

**lift-definition** *ps-states*  $::$  ( $'a$ ,  $'b$ ) *ta*  $\Rightarrow$   $'a$  *FSet-Lex-Wrapper* *fset* **is** *ps-states-set*  
*<proof>*

**lemma** *ps-states*: *ps-states*  $\mathcal{A} \subseteq$  *Wrapp*  $|\cdot|^{\dagger}$  *fPow* ( $\mathcal{Q}$   $\mathcal{A}$ ) *<proof>*

**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*  $\mathcal{A}$ )  
*<proof>*

**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>*

**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* = *TA-rule f ps* (*Wrapp p*)  $\wedge$  *fset-of-list ps*  $\subseteq$   $Q$ )

$Q \wedge$   
 $p = \text{ps-reachable-states } \mathcal{A} f (\text{map ex ps}) \wedge p \neq \{\|\}$

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

**lemma** *finite-ps-rulesp* [*simp*]:  
 $\text{finite } (\text{Collect } (\text{ps-rulesp } \mathcal{A} Q))$  (**is** *finite* ?*S*)  
 $\langle \text{proof} \rangle$

**lemmas** *finite-ps-rulesp-unfolded* = *finite-ps-rulesp*[*unfolded ps-rulesp-def*, *simplified*]

**lemmas** *ps-rulesI* [*intro!*] = *fCollect-memberI*[*OF finite-ps-rulesp*]

**lemma** *ps-rules-states*:  
 $\text{rule-states } (\text{fCollect } (\text{ps-rulesp } \mathcal{A} Q)) \mid \subseteq \mid (\text{Wrapp } \mid \mid \text{fPow } (\mathcal{Q} \mathcal{A}) \mid \cup \mid Q)$   
 $\langle \text{proof} \rangle$

**definition** *ps-ta* :: (*'q*, *'f*) *ta*  $\Rightarrow$  (*'q* *FSet-Lex-Wrapper*, *'f*) *ta* **where**  
 $\text{ps-ta } \mathcal{A} = (\text{let } Q = \text{ps-states } \mathcal{A} \text{ in}$   
 $\text{TA } (\text{ps-rules } \mathcal{A} Q) \{\|\})$

**definition** *ps-ta-Q<sub>f</sub>* :: *'q* *fset*  $\Rightarrow$  (*'q*, *'f*) *ta*  $\Rightarrow$  *'q* *FSet-Lex-Wrapper* *fset* **where**  
 $\text{ps-ta-Q}_f Q \mathcal{A} = (\text{let } Q' = \text{ps-states } \mathcal{A} \text{ in}$   
 $\text{ffilter } (\lambda S. Q \mid \cap \mid (\text{ex } S) \neq \{\|\}) Q')$

**lemma** *ps-rules-sound*:  
**assumes**  $p \mid \in \mid \text{ta-der } (\text{ps-ta } \mathcal{A})$  (*term-of-gterm* *t*)  
**shows**  $\text{ex } p \mid \subseteq \mid \text{ta-der } \mathcal{A}$  (*term-of-gterm* *t*)  $\langle \text{proof} \rangle$

**lemma** *ps-ta-nt-empty-set*:  
 $\text{TA-rule } f \text{ qs } (\text{Wrapp } \{\|\}) \mid \in \mid \text{rules } (\text{ps-ta } \mathcal{A}) \Longrightarrow \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *ps-rules-not-empty-reach*:  
**assumes**  $\text{Wrapp } \{\|\} \mid \in \mid \text{ta-der } (\text{ps-ta } \mathcal{A})$  (*term-of-gterm* *t*)  
**shows** *False*  $\langle \text{proof} \rangle$

**lemma** *ps-rules-complete*:  
**assumes**  $q \mid \in \mid \text{ta-der } \mathcal{A}$  (*term-of-gterm* *t*)  
**shows**  $\exists p. q \mid \in \mid \text{ex } p \wedge p \mid \in \mid \text{ta-der } (\text{ps-ta } \mathcal{A})$  (*term-of-gterm* *t*)  $\wedge p \mid \in \mid \text{ps-states } \mathcal{A}$   $\langle \text{proof} \rangle$

**lemma** *ps-ta-eps*[*simp*]:  $\text{eps } (\text{ps-ta } \mathcal{A}) = \{\|\}$   $\langle \text{proof} \rangle$

**lemma** *ps-ta-det*[*iff*]:  $\text{ta-det } (\text{ps-ta } \mathcal{A})$   $\langle \text{proof} \rangle$

**lemma** *ps-gta-lang*:

$gta\text{-}lang (ps\text{-}ta\text{-}Q_f Q \mathcal{A}) (ps\text{-}ta \mathcal{A}) = gta\text{-}lang Q \mathcal{A} \text{ (is } ?R = ?L)$   
 ⟨proof⟩

**definition** *ps-reg* **where**

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

**lemma** *L-ps-reg*:

$\mathcal{L} (ps\text{-}reg R) = \mathcal{L} R$

⟨proof⟩

**lemma** *ps-ta-states*:  $\mathcal{Q} (ps\text{-}ta \mathcal{A}) \mid \subseteq \mid Wrapp \mid \uparrow fPow (\mathcal{Q} \mathcal{A})$

⟨proof⟩

**lemma** *ps-ta-states'*:  $ex \mid \uparrow \mathcal{Q} (ps\text{-}ta \mathcal{A}) \mid \subseteq \mid fPow (\mathcal{Q} \mathcal{A})$

⟨proof⟩

**end**

**theory** *Tree-Automata-Complement*

**imports** *Tree-Automata-Det*

**begin**

### 3.3 Complement closure of regular languages

**definition** *partially-completely-defined-on* **where**

$partially\text{-}completely\text{-}defined\text{-}on \mathcal{A} \mathcal{F} \longleftrightarrow$

$(\forall t. funas\text{-}gterm t \subseteq fset \mathcal{F} \longleftrightarrow (\exists q. q \mid \in \mid ta\text{-}der \mathcal{A} (term\text{-}of\text{-}gterm t)))$

**definition** *sig-ta* **where**

$sig\text{-}ta \mathcal{F} = TA ((\lambda (f, n). TA\text{-}rule f (replicate n ()) ()) \mid \uparrow \mathcal{F}) \{\mid\mid\}$

**lemma** *sig-ta-rules-fmember*:

$TA\text{-}rule f qs q \mid \in \mid rules (sig\text{-}ta \mathcal{F}) \longleftrightarrow (\exists n. (f, n) \mid \in \mid \mathcal{F} \wedge qs = replicate n () \wedge q = ())$

⟨proof⟩

**lemma** *sig-ta-completely-defined*:

$partially\text{-}completely\text{-}defined\text{-}on (sig\text{-}ta \mathcal{F}) \mathcal{F}$

⟨proof⟩

**lemma** *ta-der-fsubset-sig-ta-completely*:

**assumes**  $ta\text{-}subset (sig\text{-}ta \mathcal{F}) \mathcal{A} ta\text{-}sig \mathcal{A} \mid \subseteq \mid \mathcal{F}$

**shows**  $partially\text{-}completely\text{-}defined\text{-}on \mathcal{A} \mathcal{F}$

⟨proof⟩

**lemma** *completely-defined-ps-taI*:

$partially\text{-}completely\text{-}defined\text{-}on \mathcal{A} \mathcal{F} \implies partially\text{-}completely\text{-}defined\text{-}on (ps\text{-}ta \mathcal{A}) \mathcal{F}$

⟨proof⟩

**lemma** *completely-defined-ta-unionII:*

*partially-completely-defined-on*  $\mathcal{A} \mathcal{F} \implies \text{ta-sig } \mathcal{B} \mid \subseteq \mid \mathcal{F} \implies \mathcal{Q} \mathcal{A} \mid \cap \mid \mathcal{Q} \mathcal{B} = \{\mid\}$   
 $\implies$   
*partially-completely-defined-on*  $(\text{ta-union } \mathcal{A} \mathcal{B}) \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *completely-defined-fmaps-statesI:*

*partially-completely-defined-on*  $\mathcal{A} \mathcal{F} \implies \text{finj-on } f (\mathcal{Q} \mathcal{A}) \implies \text{partially-completely-defined-on}$   
 $(\text{fmap-states-ta } f \mathcal{A}) \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *det-completely-defined-complement:*

**assumes** *partially-completely-defined-on*  $\mathcal{A} \mathcal{F}$  *ta-det*  $\mathcal{A}$   
**shows** *gta-lang*  $(\mathcal{Q} \mathcal{A} \mid - \mid \mathcal{Q}) \mathcal{A} = \mathcal{T}_G (\text{fset } \mathcal{F}) - \text{gta-lang } \mathcal{Q} \mathcal{A}$  (**is**  $?Ls = ?Rs$ )  
 $\langle \text{proof} \rangle$

**lemma** *ta-der-gterm-sig-fset:*

$q \mid \in \mid \text{ta-der } \mathcal{A} (\text{term-of-gterm } t) \implies \text{funas-gterm } t \subseteq \text{fset } (\text{ta-sig } \mathcal{A})$   
 $\langle \text{proof} \rangle$

**definition** *filter-ta-sig where*

*filter-ta-sig*  $\mathcal{F} \mathcal{A} = \text{TA } (\text{ffilter } (\lambda r. (r\text{-root } r, \text{length } (r\text{-lhs-states } r))) \mid \in \mid \mathcal{F}) (\text{rules } \mathcal{A})) (\text{eps } \mathcal{A})$

**definition** *filter-ta-reg where*

*filter-ta-reg*  $\mathcal{F} R = \text{Reg } (\text{fin } R) (\text{filter-ta-sig } \mathcal{F} (\text{ta } R))$

**lemma** *filter-ta-sig:*

*ta-sig*  $(\text{filter-ta-sig } \mathcal{F} \mathcal{A}) \mid \subseteq \mid \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *filter-ta-sig-lang:*

*gta-lang*  $\mathcal{Q} (\text{filter-ta-sig } \mathcal{F} \mathcal{A}) = \text{gta-lang } \mathcal{Q} \mathcal{A} \cap \mathcal{T}_G (\text{fset } \mathcal{F})$  (**is**  $?Ls = ?Rs$ )  
 $\langle \text{proof} \rangle$

**lemma** *L-filter-ta-reg:*

$\mathcal{L} (\text{filter-ta-reg } \mathcal{F} \mathcal{A}) = \mathcal{L} \mathcal{A} \cap \mathcal{T}_G (\text{fset } \mathcal{F})$   
 $\langle \text{proof} \rangle$

**definition** *sig-ta-reg where*

*sig-ta-reg*  $\mathcal{F} = \text{Reg } \{\mid\} (\text{sig-ta } \mathcal{F})$

**lemma** *L-sig-ta-reg:*

$\mathcal{L} (\text{sig-ta-reg } \mathcal{F}) = \{\}$   
 $\langle \text{proof} \rangle$

**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**  *$\mathcal{L}$ -complement-reg*:  
**assumes** *ta-sig* (*ta*  $\mathcal{A}$ )  $|\subseteq|$   $\mathcal{F}$   
**shows**  $\mathcal{L}$  (*complement-reg*  $\mathcal{A}$   $\mathcal{F}$ ) =  $\mathcal{T}_G$  (*fset*  $\mathcal{F}$ ) -  $\mathcal{L}$   $\mathcal{A}$   
 $\langle$ *proof* $\rangle$

**lemma**  *$\mathcal{L}$ -complement-filter-reg*:  
 $\mathcal{L}$  (*complement-reg* (*filter-ta-reg*  $\mathcal{F}$   $\mathcal{A}$ )  $\mathcal{F}$ ) =  $\mathcal{T}_G$  (*fset*  $\mathcal{F}$ ) -  $\mathcal{L}$   $\mathcal{A}$   
 $\langle$ *proof* $\rangle$

**definition** *difference-reg where*  
*difference-reg*  $R$   $L$  = (*let*  $F$  = *ta-sig* (*ta*  $R$ ) *in*  
*reg-intersect*  $R$  (*trim-reg* (*complement-reg* (*filter-ta-reg*  $F$   $L$ )  $F$ )))

**lemma**  *$\mathcal{L}$ -difference-reg*:  
 $\mathcal{L}$  (*difference-reg*  $R$   $L$ ) =  $\mathcal{L}$   $R$  -  $\mathcal{L}$   $L$  (**is** ? $L$ s = ? $R$ s)  
 $\langle$ *proof* $\rangle$

**end**  
**theory** *Tree-Automata-Pumping*  
**imports** *Tree-Automata*  
**begin**

### 3.4 Pumping lemma

**abbreviation** *derivation-ctxt*  $ts$   $Cs$   $\equiv$  *Suc* (*length*  $Cs$ ) = *length*  $ts$   $\wedge$   
 $(\forall i < \text{length } Cs. (Cs ! i) \langle ts ! i \rangle = ts ! \text{Suc } i)$

**abbreviation** *derivation-ctxt-st*  $A$   $ts$   $Cs$   $qs$   $\equiv$  *length*  $qs$  = *length*  $ts$   $\wedge$  *Suc* (*length*  $Cs$ ) = *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*  $A$   $ts$   $qs$   $\equiv$  *length*  $qs$  = *length*  $ts$   $\wedge$   
 $(\forall i < \text{length } qs. qs ! i \in | \text{ta-der } A (ts ! i))$

**definition** *derivation*  $A$   $ts$   $Cs$   $qs$   $\longleftrightarrow$  *derivation-ctxt*  $ts$   $Cs$   $\wedge$   
*derivation-ctxt-st*  $A$   $ts$   $Cs$   $qs$   $\wedge$  *derivation-sound*  $A$   $ts$   $qs$

**lemma** *ctxt-comp-lhs-not-hole*:  
**assumes**  $C \neq \square$   
**shows**  $C \circ_c D \neq \square$   
 $\langle$ *proof* $\rangle$

**lemma** *ctxt-comp-rhs-not-hole*:  
**assumes**  $D \neq \square$   
**shows**  $C \circ_c D \neq \square$   
 $\langle$ *proof* $\rangle$

**lemma** *fold-ctxt-comp-nt-empty-acc*:

**assumes**  $D \neq \square$

**shows**  $\text{fold } (\circ_c) \ Cs \ D \neq \square$

*<proof>*

**lemma** *fold-ctxt-comp-nt-empty*:

**assumes**  $C \in \text{set } Cs$  **and**  $C \neq \square$

**shows**  $\text{fold } (\circ_c) \ Cs \ D \neq \square$  *<proof>*

**lemma** *empty-ctxt-power [simp]*:

$\square \wedge n = \square$

*<proof>*

**lemma** *ctxt-comp-not-hole*:

**assumes**  $C \neq \square$  **and**  $n \neq 0$

**shows**  $C \wedge n \neq \square$

*<proof>*

**lemma** *ctxt-comp-n-suc [simp]*:

**shows**  $(C \wedge (\text{Suc } n)) \langle t \rangle = (C \wedge n) \langle C \langle t \rangle \rangle$

*<proof>*

**lemma** *ctxt-comp-reach*:

**assumes**  $p \in | \text{ta-der } A \ C \langle \text{Var } p \rangle$

**shows**  $p \in | \text{ta-der } A \ (C \wedge n) \langle \text{Var } p \rangle$

*<proof>*

**lemma** *args-depth-less [simp]*:

**assumes**  $u \in \text{set } ss$

**shows**  $\text{depth } u < \text{depth } (\text{Fun } f \ ss)$  *<proof>*

**lemma** *subterm-depth-less*:

**assumes**  $s \triangleright t$

**shows**  $\text{depth } t < \text{depth } s$

*<proof>*

**lemma** *poss-length-depth*:

**shows**  $\exists p \in \text{poss } t. \text{length } p = \text{depth } t$

*<proof>*

**lemma** *poss-length-bounded-by-depth*:

**assumes**  $p \in \text{poss } t$

**shows**  $\text{length } p \leq \text{depth } t$  *<proof>*

**lemma** *depth-ctxt-nt-hole-inc*:

**assumes**  $C \neq \square$

**shows**  $\text{depth } t < \text{depth } C\langle t \rangle$  *<proof>*

**lemma** *depth-ctxt-less-eq*:

$\text{depth } t \leq \text{depth } C\langle t \rangle$  *<proof>*

**lemma** *ctxt-comp-n-not-hole-depth-inc*:

**assumes**  $C \neq \square$

**shows**  $\text{depth } (C\hat{\ }n)\langle t \rangle < \text{depth } (C\hat{\ }(\text{Suc } n))\langle t \rangle$   
*<proof>*

**lemma** *ctxt-comp-n-lower-bound*:

**assumes**  $C \neq \square$

**shows**  $n < \text{depth } (C\hat{\ }(\text{Suc } n))\langle t \rangle$   
*<proof>*

**lemma** *ta-der-ctxt-n-loop*:

**assumes**  $q \in | \text{ta-der } \mathcal{A} \ t \ q \in | \text{ta-der } \mathcal{A} \ C\langle \text{Var } q \rangle$

**shows**  $q \in | \text{ta-der } \mathcal{A} \ (C\hat{\ }n)\langle t \rangle$   
*<proof>*

**lemma** *ctxt-compose-funs-ctxt [simp]*:

$\text{funs-ctxt } (C \circ_c D) = \text{funs-ctxt } C \cup \text{funs-ctxt } D$   
*<proof>*

**lemma** *ctxt-compose-vars-ctxt [simp]*:

$\text{vars-ctxt } (C \circ_c D) = \text{vars-ctxt } C \cup \text{vars-ctxt } D$   
*<proof>*

**lemma** *ctxt-power-funs-vars-0 [simp]*:

**assumes**  $n = 0$

**shows**  $\text{funs-ctxt } (C\hat{\ }n) = \{\}$   $\text{vars-ctxt } (C\hat{\ }n) = \{\}$   
*<proof>*

**lemma** *ctxt-power-funs-vars-n [simp]*:

**assumes**  $n \neq 0$

**shows**  $\text{funs-ctxt } (C\hat{\ }n) = \text{funs-ctxt } C$   $\text{vars-ctxt } (C\hat{\ }n) = \text{vars-ctxt } C$   
*<proof>*

**fun** *terms-pos where*

$\text{terms-pos } s \ [] = [s]$

|  $\text{terms-pos } s \ (p \# ps) = \text{terms-pos } (s \ |- \ [p]) \ ps \ @ \ [s]$

**lemma** *subt-at-poss* [simp]:

assumes  $a \# p \in \text{poss } s$

shows  $p \in \text{poss } (s \text{ |- } [a])$

$\langle \text{proof} \rangle$

**lemma** *terms-pos-length* [simp]:

shows  $\text{length } (\text{terms-pos } t \ p) = \text{Suc } (\text{length } p)$

$\langle \text{proof} \rangle$

**lemma** *terms-pos-last* [simp]:

assumes  $i = \text{length } p$

shows  $\text{terms-pos } t \ p \ ! \ i = t \ \langle \text{proof} \rangle$

**lemma** *terms-pos-subterm*:

assumes  $p \in \text{poss } t$  and  $s \in \text{set } (\text{terms-pos } t \ p)$

shows  $t \sqsupseteq s \ \langle \text{proof} \rangle$

**lemma** *terms-pos-differ-subterm*:

assumes  $p \in \text{poss } t$  and  $i < \text{length } (\text{terms-pos } t \ p)$

and  $j < \text{length } (\text{terms-pos } t \ p)$  and  $i < j$

shows  $\text{terms-pos } t \ p \ ! \ i \triangleleft \text{terms-pos } t \ p \ ! \ j$

$\langle \text{proof} \rangle$

**lemma** *distinct-terms-pos*:

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

shows  $\text{distinct } (\text{terms-pos } t \ p) \ \langle \text{proof} \rangle$

**lemma** *term-chain-depth*:

assumes  $\text{depth } t = n$

shows  $\exists p \in \text{poss } t. \text{length } (\text{terms-pos } t \ p) = (n + 1)$

$\langle \text{proof} \rangle$

**lemma** *ta-der-derivation-chain-terms-pos-exist*:

assumes  $p \in \text{poss } t$  and  $q \in | \text{ta-der } A \ t$

shows  $\exists Cs \ qs. \text{derivation } A \ (\text{terms-pos } t \ p) \ Cs \ qs \wedge \text{last } qs = q$

$\langle \text{proof} \rangle$

**lemma** *derivation-ctxt-terms-pos-nt-empty*:

assumes  $p \in \text{poss } t$  and  $\text{derivation-ctxt } (\text{terms-pos } t \ p) \ Cs$  and  $C \in \text{set } Cs$

shows  $C \neq \square$

$\langle \text{proof} \rangle$

**lemma** *derivation-ctxt-terms-pos-sub-list-nt-empty*:

assumes  $p \in \text{poss } t$  and  $\text{derivation-ctxt } (\text{terms-pos } t \ p) \ Cs$

and  $i < \text{length } Cs$  and  $j \leq \text{length } Cs$  and  $i < j$

shows  $\text{fold } (\circ_c) \ (\text{take } (j - i) \ (\text{drop } i \ Cs)) \ \square \neq \square$

$\langle \text{proof} \rangle$

**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**  $(\text{fold } (\circ_c) (\text{take } (j - i) (\text{drop } i Cs)) \square) \langle ts ! i \rangle = ts ! j$

*<proof>*

**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**  $qs ! j \in | \text{ta-der } A (\text{fold } (\circ_c) (\text{take } (j - i) (\text{drop } i Cs)) \square) \langle \text{Var } (qs ! i) \rangle$

*<proof>*

**lemma** *terms-pos-ground*:

**assumes** *ground t and*  $p \in \text{poss } t$

**shows**  $\forall s \in \text{set } (\text{terms-pos } t p). \text{ground } s$

*<proof>*

**lemma** *list-card-smaller-contains-eq-elemens*:

**assumes**  $\text{length } qs = n$  **and**  $\text{card } (\text{set } qs) < n$

**shows**  $\exists i < \text{length } qs. \exists j < \text{length } qs. i < j \wedge qs ! i = qs ! j$

*<proof>*

**lemma** *length-remdups-less-eq*:

**assumes**  $\text{set } xs \subseteq \text{set } ys$

**shows**  $\text{length } (\text{remdups } xs) \leq \text{length } (\text{remdups } ys)$  *<proof>*

**lemma** *pigeonhole-tree-automata*:

**assumes**  $\text{fcard } (\mathcal{Q} A) < \text{depth } t$  **and**  $q \in | \text{ta-der } A t$  **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>*

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

**shows**  $\text{ta-der } \mathcal{A} (\text{term-of-gterm } C \langle s \rangle_G) = \text{ta-der } \mathcal{A} (\text{term-of-gterm } C \langle t \rangle_G)$  *<proof>*

**locale** *myhill-nerode* =

**fixes**  $\mathcal{F} \mathcal{L}$  **assumes** *term-subset*:  $\mathcal{L} \subseteq \mathcal{T}_G \mathcal{F}$

**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*:  $s \equiv_{\mathcal{L}} t \implies s \in \mathcal{T}_G\ \mathcal{F} \quad s \equiv_{\mathcal{L}} t \implies t \in \mathcal{T}_G\ \mathcal{F}$   
*<proof>*

**lemma** *myhill-refl* [*simp*]:  $s \in \mathcal{T}_G\ \mathcal{F} \implies s \equiv_{\mathcal{L}} s$   
*<proof>*

**lemma** *myhill-symmetric*:  $s \equiv_{\mathcal{L}} t \implies t \equiv_{\mathcal{L}} s$   
*<proof>*

**lemma** *myhill-trans* [*trans*]:  
 $s \equiv_{\mathcal{L}} t \implies t \equiv_{\mathcal{L}} u \implies s \equiv_{\mathcal{L}} u$   
*<proof>*

**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* ( $\mathcal{T}_G\ \mathcal{F}$ )  $MN_{\mathcal{L}}$   
*<proof>*

**lemma** *rtl-der-image-on-myhill-inj*:  
**assumes** *gta-lang*  $Q_f\ \mathcal{A} = \mathcal{L}$   
**shows** *inj-on* ( $\lambda X. gta-der\ \mathcal{A}\ 'X$ ) ( $\mathcal{T}_G\ \mathcal{F} // MN_{\mathcal{L}}$ ) (**is inj-on** ?D ?R)  
*<proof>*

**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>*

**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**

$gtt-rules\ \mathcal{G} \equiv rules\ (fst\ \mathcal{G}) \mid \cup \mid rules\ (snd\ \mathcal{G})$

**abbreviation** *gtt-eps* **where**

$$gtt-eps \mathcal{G} \equiv eps (fst \mathcal{G}) \mid \cup \mid eps (snd \mathcal{G})$$

**definition** *gtt-states* **where**

$$gtt-states \mathcal{G} = \mathcal{Q} (fst \mathcal{G}) \mid \cup \mid \mathcal{Q} (snd \mathcal{G})$$

**abbreviation** *gtt-syms* **where**

$$gtt-syms \mathcal{G} \equiv ta-sig (fst \mathcal{G}) \mid \cup \mid ta-sig (snd \mathcal{G})$$

**definition** *gtt-interface* **where**

$$gtt-interface \mathcal{G} = \mathcal{Q} (fst \mathcal{G}) \mid \cap \mid \mathcal{Q} (snd \mathcal{G})$$

**definition** *gtt-eps-free* **where**

$$gtt-eps-free \mathcal{G} = (eps-free (fst \mathcal{G}), eps-free (snd \mathcal{G}))$$

**definition** *is-gtt-eps-free* :: ('q, 'f) ta × ('p, 'g) ta ⇒ bool **where**

$$is-gtt-eps-free \mathcal{G} \longleftrightarrow eps (fst \mathcal{G}) = \{\}\ \wedge\ eps (snd \mathcal{G}) = \{\}$$

\*anchored\* language accepted by a GTT

**definition** *agtt-lang* :: ('q, 'f) gtt ⇒ 'f gterm rel **where**

$$agtt-lang \mathcal{G} = \{(t, u) \mid t \ u \ q. \ q \mid \in \mid gta-der (fst \mathcal{G}) \ t \ \wedge \ q \mid \in \mid gta-der (snd \mathcal{G}) \ u\}$$

**lemma** *agtt-langI*:

$$q \mid \in \mid gta-der (fst \mathcal{G}) \ s \implies q \mid \in \mid gta-der (snd \mathcal{G}) \ t \implies (s, t) \in agtt-lang \mathcal{G}$$

⟨proof⟩

**lemma** *agtt-langE*:

**assumes** (s, t) ∈ agtt-lang  $\mathcal{G}$

**obtains** q **where** q ∈ gta-der (fst  $\mathcal{G}$ ) s q ∈ gta-der (snd  $\mathcal{G}$ ) t

⟨proof⟩

**lemma** *converse-agtt-lang*:

$$(agtt-lang \mathcal{G})^{-1} = agtt-lang (prod.swap \mathcal{G})$$

⟨proof⟩

**lemma** *agtt-lang-swap*:

$$agtt-lang (prod.swap \mathcal{G}) = prod.swap \ ` agtt-lang \mathcal{G}$$

⟨proof⟩

language accepted by a GTT

**abbreviation** *gtt-lang* :: ('q, 'f) gtt ⇒ 'f gterm rel **where**

$$gtt-lang \mathcal{G} \equiv gmctxt-cl UNIV (agtt-lang \mathcal{G})$$

**lemma** *gtt-lang-join*:

$$q \mid \in \mid gta-der (fst \mathcal{G}) \ s \implies q \mid \in \mid gta-der (snd \mathcal{G}) \ t \implies (s, t) \in gmctxt-cl UNIV (agtt-lang \mathcal{G})$$

⟨proof⟩

**definition** *gtt-accept* **where**

$$gtt-accept \mathcal{G} \ s \ t \equiv (s, t) \in gmctxt-cl UNIV (agtt-lang \mathcal{G})$$

**lemma** *gtt-accept-intros*:

$$(s, t) \in agtt-lang \mathcal{G} \implies gtt-accept \mathcal{G} \ s \ t$$

$length\ ss = length\ ts \implies \forall i < length\ ts. gtt\text{-}accept\ \mathcal{G}\ (ss\ !\ i)\ (ts\ !\ i) \implies$   
 $(f, length\ ss) \in \mathcal{F} \implies gtt\text{-}accept\ \mathcal{G}\ (GFun\ f\ ss)\ (GFun\ f\ ts)$   
 ⟨proof⟩

**abbreviation**  $gtt\text{-}lang\text{-}terms :: ('q, 'f) gtt \Rightarrow ('f, 'q) term\ rel$  **where**  
 $gtt\text{-}lang\text{-}terms\ \mathcal{G} \equiv (\lambda s. map\text{-}both\ term\text{-}of\text{-}gterm\ s)\ ' (gmctxt\text{-}cl\ UNIV\ (agtt\text{-}lang\ \mathcal{G}))$

**lemma**  $term\text{-}of\text{-}gterm\text{-}gtt\text{-}lang\text{-}gtt\text{-}lang\text{-}terms\text{-}conv$ :  
 $map\text{-}both\ term\text{-}of\text{-}gterm\ ' gtt\text{-}lang\ \mathcal{G} = gtt\text{-}lang\text{-}terms\ \mathcal{G}$   
 ⟨proof⟩

**lemma**  $gtt\text{-}accept\text{-}swap$  [simp]:  
 $gtt\text{-}accept\ (prod.swap\ \mathcal{G})\ s\ t \longleftrightarrow gtt\text{-}accept\ \mathcal{G}\ t\ s$   
 ⟨proof⟩

**lemma**  $gtt\text{-}lang\text{-}swap$ :  
 $(gtt\text{-}lang\ (A, B))^{-1} = gtt\text{-}lang\ (B, A)$   
 ⟨proof⟩

**lemma**  $gtt\text{-}accept\text{-}exI$ :  
**assumes**  $gtt\text{-}accept\ \mathcal{G}\ s\ t$   
**shows**  $\exists u. u \in ta\text{-}der'\ (fst\ \mathcal{G})\ (term\text{-}of\text{-}gterm\ s) \wedge u \in ta\text{-}der'\ (snd\ \mathcal{G})\ (term\text{-}of\text{-}gterm\ t)$   
 ⟨proof⟩

**lemma**  $agtt\text{-}lang\text{-}mono$ :  
**assumes**  $rules\ (fst\ \mathcal{G}) \subseteq rules\ (fst\ \mathcal{G}')\ eps\ (fst\ \mathcal{G}) \subseteq eps\ (fst\ \mathcal{G}')$   
 $rules\ (snd\ \mathcal{G}) \subseteq rules\ (snd\ \mathcal{G}')\ eps\ (snd\ \mathcal{G}) \subseteq eps\ (snd\ \mathcal{G}')$   
**shows**  $agtt\text{-}lang\ \mathcal{G} \subseteq agtt\text{-}lang\ \mathcal{G}'$   
 ⟨proof⟩

**lemma**  $gtt\text{-}lang\text{-}mono$ :  
**assumes**  $rules\ (fst\ \mathcal{G}) \subseteq rules\ (fst\ \mathcal{G}')\ eps\ (fst\ \mathcal{G}) \subseteq eps\ (fst\ \mathcal{G}')$   
 $rules\ (snd\ \mathcal{G}) \subseteq rules\ (snd\ \mathcal{G}')\ eps\ (snd\ \mathcal{G}) \subseteq eps\ (snd\ \mathcal{G}')$   
**shows**  $gtt\text{-}lang\ \mathcal{G} \subseteq gtt\text{-}lang\ \mathcal{G}'$   
 ⟨proof⟩

**definition**  $fmap\text{-}states\text{-}gtt$  **where**  
 $fmap\text{-}states\text{-}gtt\ f \equiv map\text{-}both\ (fmap\text{-}states\text{-}ta\ f)$

**lemma**  $ground\text{-}map\text{-}vars\text{-}term\text{-}simp$ :  
 $ground\ t \implies map\text{-}term\ f\ g\ t = map\text{-}term\ f\ (\lambda\_. undefined)\ t$   
 ⟨proof⟩

**lemma**  $states\text{-}fmap\text{-}states\text{-}gtt$  [simp]:

$gtt\text{-states } (fmap\text{-states-gtt } f \mathcal{G}) = f \mid^{\cdot} gtt\text{-states } \mathcal{G}$   
 ⟨proof⟩

**lemma** *agtt-lang-fmap-states-gtt*:  
 assumes *finj-on f (gtt-states  $\mathcal{G}$ )*  
 shows  $agtt\text{-lang } (fmap\text{-states-gtt } f \mathcal{G}) = agtt\text{-lang } \mathcal{G}$  (is ?Ls = ?Rs)  
 ⟨proof⟩

**lemma** *agtt-lang-Inl-Inr-states-agtt*:  
 $agtt\text{-lang } (fmap\text{-states-gtt } Inl \mathcal{G}) = agtt\text{-lang } \mathcal{G}$   
 $agtt\text{-lang } (fmap\text{-states-gtt } Inr \mathcal{G}) = agtt\text{-lang } \mathcal{G}$   
 ⟨proof⟩

**lemma** *gtt-lang-fmap-states-gtt*:  
 assumes *finj-on f (gtt-states  $\mathcal{G}$ )*  
 shows  $gtt\text{-lang } (fmap\text{-states-gtt } f \mathcal{G}) = gtt\text{-lang } \mathcal{G}$  (is ?Ls = ?Rs)  
 ⟨proof⟩

**definition** *gtt-only-reach* where  
 $gtt\text{-only-reach} = map\text{-both } ta\text{-only-reach}$

#### 4.1 (A)GTT reachable states

**lemma** *agtt-only-reach-lang*:  
 $agtt\text{-lang } (gtt\text{-only-reach } \mathcal{G}) = agtt\text{-lang } \mathcal{G}$   
 ⟨proof⟩

**lemma** *gtt-only-reach-lang*:  
 $gtt\text{-lang } (gtt\text{-only-reach } \mathcal{G}) = gtt\text{-lang } \mathcal{G}$   
 ⟨proof⟩

**lemma** *gtt-only-reach-syms*:  
 $gtt\text{-syms } (gtt\text{-only-reach } \mathcal{G}) \mid\subseteq\mid gtt\text{-syms } \mathcal{G}$   
 ⟨proof⟩

#### 4.2 (A)GTT productive states

**definition** *gtt-only-prod* where  
 $gtt\text{-only-prod } \mathcal{G} = (let \text{iface} = gtt\text{-interface } \mathcal{G} \text{ in}$   
 $map\text{-both } (ta\text{-only-prod } \text{iface}) \mathcal{G})$

**lemma** *agtt-only-prod-lang*:  
 $agtt\text{-lang } (gtt\text{-only-prod } \mathcal{G}) = agtt\text{-lang } \mathcal{G}$  (is ?Ls = ?Rs)  
 ⟨proof⟩

**lemma** *gtt-only-prod-lang*:  
 $gtt\text{-lang } (gtt\text{-only-prod } \mathcal{G}) = gtt\text{-lang } \mathcal{G}$   
 ⟨proof⟩

**lemma** *gtt-only-prod-syms*:

$gtt\text{-syms } (gtt\text{-only-prod } \mathcal{G}) \mid \subseteq \mid gtt\text{-syms } \mathcal{G}$   
 $\langle \text{proof} \rangle$

### 4.3 (A)GTT trimming

**definition** *trim-gtt* **where**

$$\text{trim-gtt} = gtt\text{-only-prod} \circ gtt\text{-only-reach}$$

**lemma** *trim-agtt-lang*:

$$agtt\text{-lang } (\text{trim-gtt } G) = agtt\text{-lang } G$$

$\langle \text{proof} \rangle$

**lemma** *trim-gtt-lang*:

$$gtt\text{-lang } (\text{trim-gtt } G) = gtt\text{-lang } G$$

$\langle \text{proof} \rangle$

**lemma** *trim-gtt-prod-syms*:

$$gtt\text{-syms } (\text{trim-gtt } G) \mid \subseteq \mid gtt\text{-syms } G$$

$\langle \text{proof} \rangle$

### 4.4 root-cleanliness

A GTT is root-clean if none of its interface states can occur in a non-root positions in the accepting derivations corresponding to its anchored GTT relation.

**definition** *ta-nr-states*  $:: ('q, 'f) \text{ ta} \Rightarrow 'q \text{ fset}$  **where**

$$\text{ta-nr-states } A = \mid \cup \mid ((\text{fset-of-list} \circ r\text{-lhs-states}) \mid ' \mid (\text{rules } A))$$

**definition** *gtt-nr-states* **where**

$$gtt\text{-nr-states } G = \text{ta-nr-states } (\text{fst } G) \mid \cup \mid \text{ta-nr-states } (\text{snd } G)$$

**definition** *gtt-root-clean* **where**

$$gtt\text{-root-clean } G \longleftrightarrow gtt\text{-nr-states } G \mid \cap \mid gtt\text{-interface } G = \{\mid\}$$

### 4.5 Relabeling

**definition** *relabel-gtt*  $:: ('q :: \text{linorder}, 'f) \text{ gtt} \Rightarrow (\text{nat}, 'f) \text{ gtt}$  **where**

$$\text{relabel-gtt } G = \text{fmap-states-gtt } (\text{map-fset-to-nat } (gtt\text{-states } G)) G$$

**lemma** *relabel-agtt-lang* [*simp*]:

$$agtt\text{-lang } (\text{relabel-gtt } G) = agtt\text{-lang } G$$

$\langle \text{proof} \rangle$

**lemma** *agtt-lang-sig*:

$$\text{fset } (gtt\text{-syms } G) \subseteq \mathcal{F} \Longrightarrow agtt\text{-lang } G \subseteq \mathcal{T}_G \mathcal{F} \times \mathcal{T}_G \mathcal{F}$$

$\langle \text{proof} \rangle$

## 4.6 epsilon free GTTs

**lemma** *agtt-lang-gtt-eps-free* [simp]:  
 $agtt\text{-}lang\ (gtt\text{-}eps\text{-}free\ \mathcal{G}) = agtt\text{-}lang\ \mathcal{G}$   
 ⟨proof⟩

**lemma** *gtt-lang-gtt-eps-free* [simp]:  
 $gtt\text{-}lang\ (gtt\text{-}eps\text{-}free\ \mathcal{G}) = gtt\text{-}lang\ \mathcal{G}$   
 ⟨proof⟩

**end**  
**theory** *GTT-Compose*  
 imports *GTT*  
**begin**

## 4.7 GTT closure under composition

**inductive-set**  $\Delta_\varepsilon\text{-set} :: ('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta \Rightarrow ('q \times 'q)\ set$  **for**  $\mathcal{A}\ \mathcal{B}$  **where**  
 $\Delta_\varepsilon\text{-set}\text{-cong}: TA\text{-rule}\ f\ ps\ p\ |\in| rules\ \mathcal{A} \Longrightarrow TA\text{-rule}\ f\ qs\ q\ |\in| rules\ \mathcal{B} \Longrightarrow length\ ps = length\ qs \Longrightarrow$   
 $(\bigwedge i. i < length\ qs \Longrightarrow (ps\ !\ i, qs\ !\ i) \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B}) \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B}$   
 $|\ \Delta_\varepsilon\text{-set}\text{-eps1}: (p, p')\ |\in| eps\ \mathcal{A} \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p', q) \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B}$   
 $|\ \Delta_\varepsilon\text{-set}\text{-eps2}: (q, q')\ |\in| eps\ \mathcal{B} \Longrightarrow (p, q) \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p, q') \in \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B}$

**lemma**  $\Delta_\varepsilon\text{-states}: \Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B} \subseteq fset\ (\mathcal{Q}\ \mathcal{A}\ |\times| \mathcal{Q}\ \mathcal{B})$   
 ⟨proof⟩

**lemma** *finite- $\Delta_\varepsilon$*  [simp]: *finite*  $(\Delta_\varepsilon\text{-set}\ \mathcal{A}\ \mathcal{B})$   
 ⟨proof⟩

**context**  
**includes** *fset.lifting*  
**begin**

**lift-definition**  $\Delta_\varepsilon :: ('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta \Rightarrow ('q \times 'q)\ fset$  **is**  $\Delta_\varepsilon\text{-set}$  ⟨proof⟩

**lemmas**  $\Delta_\varepsilon\text{-cong} = \Delta_\varepsilon\text{-set}\text{-cong}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-eps1} = \Delta_\varepsilon\text{-set}\text{-eps1}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-eps2} = \Delta_\varepsilon\text{-set}\text{-eps2}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-cases} = \Delta_\varepsilon\text{-set}\text{-cases}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-induct}$  [*consumes 1, case-names  $\Delta_\varepsilon\text{-cong}\ \Delta_\varepsilon\text{-eps1}\ \Delta_\varepsilon\text{-eps2}$* ] =  $\Delta_\varepsilon\text{-set}\text{-induct}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-intros} = \Delta_\varepsilon\text{-set}\text{-intros}$  [*Transfer.transferred*]

**lemmas**  $\Delta_\varepsilon\text{-simps} = \Delta_\varepsilon\text{-set}\text{-simps}$  [*Transfer.transferred*]

**end**

**lemma** *finite-alt-def* [simp]:  
 $finite\ \{(\alpha, \beta). (\exists t. ground\ t \wedge \alpha\ |\in| ta\text{-der}\ \mathcal{A}\ t \wedge \beta\ |\in| ta\text{-der}\ \mathcal{B}\ t)\}$  (**is finite ?S**)  
 ⟨proof⟩

**lemma**  $\Delta_\varepsilon\text{-def}'$ :  
 $\Delta_\varepsilon\ \mathcal{A}\ \mathcal{B} = \{(\alpha, \beta). (\exists t. ground\ t \wedge \alpha\ |\in| ta\text{-der}\ \mathcal{A}\ t \wedge \beta\ |\in| ta\text{-der}\ \mathcal{B}\ t)\}$   
 ⟨proof⟩

**lemma**  $\Delta_\varepsilon$ -fmember:

$$(p, q) \mid \in \mid \Delta_\varepsilon \mathcal{A} \mathcal{B} \longleftrightarrow (\exists t. \text{ground } t \wedge p \mid \in \mid \text{ta-der } \mathcal{A} \ t \wedge q \mid \in \mid \text{ta-der } \mathcal{B} \ t)$$

*<proof>*

**definition**  $GTT\text{-comp} :: ('q, 'f) \text{gtt} \Rightarrow ('q, 'f) \text{gtt} \Rightarrow ('q, 'f) \text{gtt}$  **where**

$$\begin{aligned} &GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2 = \\ &(\text{let } \Delta = \Delta_\varepsilon (\text{snd } \mathcal{G}_1) (\text{fst } \mathcal{G}_2) \text{ in} \\ &(\text{TA } (\text{gtt-rules } (\text{fst } \mathcal{G}_1, \text{fst } \mathcal{G}_2)) (\text{eps } (\text{fst } \mathcal{G}_1) \mid \cup \mid \text{eps } (\text{fst } \mathcal{G}_2) \mid \cup \mid \Delta), \\ &\text{TA } (\text{gtt-rules } (\text{snd } \mathcal{G}_1, \text{snd } \mathcal{G}_2)) (\text{eps } (\text{snd } \mathcal{G}_1) \mid \cup \mid \text{eps } (\text{snd } \mathcal{G}_2) \mid \cup \mid (\Delta \mid^{-1} \mid)))) \end{aligned}$$

**lemma**  $\text{gtt-syms-GTT-comp}$ :

$$\text{gtt-syms } (GTT\text{-comp } A \ B) = \text{gtt-syms } A \ \mid \cup \mid \text{gtt-syms } B$$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -statesD:

$$\begin{aligned} (p, q) \mid \in \mid \Delta_\varepsilon \mathcal{A} \mathcal{B} &\implies p \mid \in \mid \mathcal{Q} \ \mathcal{A} \\ (p, q) \mid \in \mid \Delta_\varepsilon \mathcal{A} \mathcal{B} &\implies q \mid \in \mid \mathcal{Q} \ \mathcal{B} \end{aligned}$$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -statesD':

$$q \mid \in \mid \text{eps-states } (\Delta_\varepsilon \mathcal{A} \ \mathcal{B}) \implies q \mid \in \mid \mathcal{Q} \ \mathcal{A} \ \mid \cup \mid \mathcal{Q} \ \mathcal{B}$$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -swap:

$$\text{prod.swap } p \mid \in \mid \Delta_\varepsilon \mathcal{A} \ \mathcal{B} \longleftrightarrow p \mid \in \mid \Delta_\varepsilon \mathcal{B} \ \mathcal{A}$$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -inverse [simp]:

$$(\Delta_\varepsilon \mathcal{A} \ \mathcal{B}) \mid^{-1} \mid = \Delta_\varepsilon \mathcal{B} \ \mathcal{A}$$

*<proof>*

**lemma**  $\text{gtt-states-comp-union}$ :

$$\text{gtt-states } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2) \mid \subseteq \mid \text{gtt-states } \mathcal{G}_1 \ \mid \cup \mid \text{gtt-states } \mathcal{G}_2$$

*<proof>*

**lemma**  $GTT\text{-comp-swap}$  [simp]:

$$GTT\text{-comp } (\text{prod.swap } \mathcal{G}_2) (\text{prod.swap } \mathcal{G}_1) = \text{prod.swap } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)$$

*<proof>*

**lemma**  $\text{gtt-comp-complete-semi}$ :

**assumes**  $s: q \mid \in \mid \text{gta-der } (\text{fst } \mathcal{G}_1) \ s$  **and**  $u: q \mid \in \mid \text{gta-der } (\text{snd } \mathcal{G}_1) \ u$  **and**  $ut: \text{gtt-accept } \mathcal{G}_2 \ u \ t$

**shows**  $q \mid \in \mid \text{gta-der } (\text{fst } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ s$   $q \mid \in \mid \text{gta-der } (\text{snd } (GTT\text{-comp } \mathcal{G}_1 \ \mathcal{G}_2)) \ t$

*<proof>*

**lemmas**  $\text{gtt-comp-complete-semi}' = \text{gtt-comp-complete-semi}$ [of -  $\text{prod.swap } \mathcal{G}_2$  - -

*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>*

**lemma**  $\Delta_\varepsilon$ -*steps-from- $\mathcal{G}_2$* :

**assumes**  $(q, q') \in | \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | q \in | gtt-states \mathcal{G}_2$   
*gtt-states  $\mathcal{G}_1 \cap | gtt-states \mathcal{G}_2 = \{|\}$*   
**shows**  $(q, q') \in | \in | (eps (fst \mathcal{G}_2))^+ | \wedge q' \in | gtt-states \mathcal{G}_2$   
*<proof>*

**lemma**  $\Delta_\varepsilon$ -*steps-from- $\mathcal{G}_1$* :

**assumes**  $(p, r) \in | \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | p \in | gtt-states \mathcal{G}_1$   
*gtt-states  $\mathcal{G}_1 \cap | gtt-states \mathcal{G}_2 = \{|\}$*   
**obtains**  $r \in | gtt-states \mathcal{G}_1 (p, r) \in | (eps (fst \mathcal{G}_1))^+ |$   
 $| q p'$  **where**  $r \in | gtt-states \mathcal{G}_2 p = p' \vee (p, p') \in | (eps (fst \mathcal{G}_1))^+ | (p', q) \in |$   
 $\Delta_\varepsilon (snd \mathcal{G}_1) (fst \mathcal{G}_2)$   
 $q = r \vee (q, r) \in | (eps (fst \mathcal{G}_2))^+ |$   
*<proof>*

**lemma**  $\Delta_\varepsilon$ -*steps-from- $\mathcal{G}_1 \mathcal{G}_2$* :

**assumes**  $(q, q') \in | \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | q \in | gtt-states \mathcal{G}_1 \cup |$   
*gtt-states  $\mathcal{G}_2$*   
*gtt-states  $\mathcal{G}_1 \cap | gtt-states \mathcal{G}_2 = \{|\}$*   
**obtains**  $q \in | gtt-states \mathcal{G}_1 q' \in | gtt-states \mathcal{G}_1 (q, q') \in | (eps (fst \mathcal{G}_1))^+ |$   
 $| p p'$  **where**  $q \in | gtt-states \mathcal{G}_1 q' \in | gtt-states \mathcal{G}_2 q = p \vee (q, p) \in | (eps (fst$   
 $\mathcal{G}_1))^+ |$   
 $(p, p') \in | \Delta_\varepsilon (snd \mathcal{G}_1) (fst \mathcal{G}_2) p' = q' \vee (p', q') \in | (eps (fst \mathcal{G}_2))^+ |$   
 $| q \in | gtt-states \mathcal{G}_2 (q, q') \in | (eps (fst \mathcal{G}_2))^+ | \wedge q' \in | gtt-states \mathcal{G}_2$   
*<proof>*

**lemma** *GTT-comp-eps-fst-statesD*:

$(p, q) \in | \in | eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) \implies p \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$   
 $(p, q) \in | \in | eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)) \implies q \in | gtt-states \mathcal{G}_1 \cup | gtt-states \mathcal{G}_2$   
*<proof>*

**lemma** *GTT-comp-eps-ftrancl-fst-statesD*:

$(p, q) \in | \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | \implies p \in | gtt-states \mathcal{G}_1 \cup | gtt-states$   
 $\mathcal{G}_2$   
 $(p, q) \in | \in | (eps (fst (GTT-comp \mathcal{G}_1 \mathcal{G}_2)))|^+ | \implies q \in | gtt-states \mathcal{G}_1 \cup | gtt-states$   
 $\mathcal{G}_2$   
*<proof>*

**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**  $q \in | ta-der (fst \mathcal{G}_1) t$

*<proof>*

**lemma** *GTT-comp-second*:

**assumes** *gtt-states*  $\mathcal{G}_1 \mid \cap \mid$  *gtt-states*  $\mathcal{G}_2 = \{\mid\}$   $q \mid \in \mid$  *gtt-states*  $\mathcal{G}_2$

$q \mid \in \mid$  *ta-der* (*snd* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ ))  $t$

**shows**  $q \mid \in \mid$  *ta-der* (*snd*  $\mathcal{G}_2$ )  $t$

*<proof>*

**lemma** *gtt-comp-sound-semi*:

**fixes**  $\mathcal{G}_1 \mathcal{G}_2 :: ('f, 'q)$  *gtt*

**assumes** *as2*: *gtt-states*  $\mathcal{G}_1 \mid \cap \mid$  *gtt-states*  $\mathcal{G}_2 = \{\mid\}$

**and**  $1: q \mid \in \mid$  *gta-der* (*fst* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ ))  $s$   $q \mid \in \mid$  *gta-der* (*snd* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ ))  $t$   $q \mid \in \mid$  *gtt-states*  $\mathcal{G}_1$

**shows**  $\exists u. q \mid \in \mid$  *gta-der* (*snd*  $\mathcal{G}_1$ )  $u \wedge$  *gtt-accept*  $\mathcal{G}_2$   $u$   $t$  *<proof>*

**lemma** *gtt-comp-asound*:

**assumes** *gtt-states*  $\mathcal{G}_1 \mid \cap \mid$  *gtt-states*  $\mathcal{G}_2 = \{\mid\}$

**shows** *agtt-lang* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ )  $\subseteq$  *gcomp-rel UNIV* (*agtt-lang*  $\mathcal{G}_1$ ) (*agtt-lang*  $\mathcal{G}_2$ )

*<proof>*

**lemma** *gtt-comp-lang-complete*:

**shows** *gtt-lang*  $\mathcal{G}_1$   $O$  *gtt-lang*  $\mathcal{G}_2 \subseteq$  *gtt-lang* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ )

*<proof>*

**lemma** *gtt-comp-alang*:

**assumes** *gtt-states*  $\mathcal{G}_1 \mid \cap \mid$  *gtt-states*  $\mathcal{G}_2 = \{\mid\}$

**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>*

**lemma** *gtt-comp-lang*:

**assumes** *gtt-states*  $\mathcal{G}_1 \mid \cap \mid$  *gtt-states*  $\mathcal{G}_2 = \{\mid\}$

**shows** *gtt-lang* (*GTT-comp*  $\mathcal{G}_1 \mathcal{G}_2$ ) = *gtt-lang*  $\mathcal{G}_1$   $O$  *gtt-lang*  $\mathcal{G}_2$

*<proof>*

**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>*

**end**

**theory** *GTT-Transitive-Closure*

**imports** *GTT-Compose*

**begin**

## 4.8 GTT closure under transitivity

**inductive-set**  $\Delta$ -trancl-set :: ('q, 'f) ta  $\Rightarrow$  ('q, 'f) ta  $\Rightarrow$  ('q  $\times$  'q) set for A B where

$\Delta$ -set-cong: TA-rule f ps p | $\in$ | rules A  $\Longrightarrow$  TA-rule f qs q | $\in$ | rules B  $\Longrightarrow$  length ps = length qs  $\Longrightarrow$

( $\bigwedge i. i < \text{length } qs \Longrightarrow (ps ! i, qs ! i) \in \Delta$ -trancl-set A B)  $\Longrightarrow (p, q) \in \Delta$ -trancl-set A B

|  $\Delta$ -set-eps1: (p, p') | $\in$ | eps A  $\Longrightarrow (p, q) \in \Delta$ -trancl-set A B  $\Longrightarrow (p', q) \in \Delta$ -trancl-set A B

|  $\Delta$ -set-eps2: (q, q') | $\in$ | eps B  $\Longrightarrow (p, q) \in \Delta$ -trancl-set A B  $\Longrightarrow (p, q') \in \Delta$ -trancl-set A B

|  $\Delta$ -set-trans: (p, q)  $\in \Delta$ -trancl-set A B  $\Longrightarrow (q, r) \in \Delta$ -trancl-set A B  $\Longrightarrow (p, r) \in \Delta$ -trancl-set A B

**lemma**  $\Delta$ -trancl-set-states:  $\Delta$ -trancl-set A B  $\subseteq$  fset (Q A | $\times$ | Q B)

*<proof>*

**lemma** finite- $\Delta$ -trancl-set [simp]: finite ( $\Delta$ -trancl-set A B)

*<proof>*

**context**

**includes** fset.lifting

**begin**

**lift-definition**  $\Delta$ -trancl :: ('q, 'f) ta  $\Rightarrow$  ('q, 'f) ta  $\Rightarrow$  ('q  $\times$  'q) fset **is**  $\Delta$ -trancl-set

*<proof>*

**lemmas**  $\Delta$ -trancl-cong =  $\Delta$ -set-cong [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-eps1 =  $\Delta$ -set-eps1 [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-eps2 =  $\Delta$ -set-eps2 [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-cases =  $\Delta$ -trancl-set.cases [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-induct [consumes 1, case-names  $\Delta$ -cong  $\Delta$ -eps1  $\Delta$ -eps2  $\Delta$ -trans] =  $\Delta$ -trancl-set.induct [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-intros =  $\Delta$ -trancl-set.intros [Transfer.transferred]

**lemmas**  $\Delta$ -trancl-simps =  $\Delta$ -trancl-set.simps [Transfer.transferred]

**end**

**lemma**  $\Delta$ -trancl-cl [simp]:

( $\Delta$ -trancl A B)<sup>+</sup> =  $\Delta$ -trancl A B

*<proof>*

**lemma**  $\Delta$ -trancl-states:  $\Delta$ -trancl A B  $\subseteq$  (Q A | $\times$ | Q B)

*<proof>*

**definition** GTT-trancl **where**

GTT-trancl G =

(let  $\Delta = \Delta$ -trancl (snd G) (fst G) in

(TA (rules (fst G)) (eps (fst G) | $\cup$ |  $\Delta$ ),

TA (rules (snd G)) (eps (snd G) | $\cup$ | ( $\Delta$ <sup>-1</sup>))))

**lemma**  $\Delta$ -trancl-inv:

$$(\Delta\text{-trancl } A \ B)^{-1} = \Delta\text{-trancl } B \ A$$

*<proof>*

**lemma** *gtt-states-GTT-trancl*:

$$\text{gtt-states } (GTT\text{-trancl } G) \mid \subseteq \mid \text{gtt-states } G$$

*<proof>*

**lemma** *gtt-syms-GTT-trancl*:

$$\text{gtt-syms } (GTT\text{-trancl } G) = \text{gtt-syms } G$$

*<proof>*

**lemma** *GTT-trancl-base*:

$$\text{gtt-lang } G \subseteq \text{gtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**lemma** *GTT-trancl-trans*:

$$\text{gtt-lang } (GTT\text{-comp } (GTT\text{-trancl } G) \ (GTT\text{-trancl } G)) \subseteq \text{gtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**lemma** *agtt-lang-base*:

$$\text{agtt-lang } G \subseteq \text{agtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -tr-incl:

$$\Delta_\varepsilon (TA \ (\text{rules } A) \ (\text{eps } A \ \mid \cup \mid \ \Delta\text{-trancl } B \ A)) \ (TA \ (\text{rules } B) \ (\text{eps } B \ \mid \cup \mid \ \Delta\text{-trancl } A \ B)) = \Delta\text{-trancl } A \ B$$

(is ?LS = ?RS)

*<proof>*

**lemma** *agtt-lang-trans*:

$$\text{gcomp-rel UNIV } (\text{agtt-lang } (GTT\text{-trancl } G)) \ (\text{agtt-lang } (GTT\text{-trancl } G)) \subseteq \text{agtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**lemma** *GTT-trancl-acomplete*:

$$\text{gtrancl-rel UNIV } (\text{agtt-lang } G) \subseteq \text{agtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**lemma** *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*:

$$(\text{gtt-lang } G)^* = (\text{gtt-lang } G)^+$$

*<proof>*

**lemma** *GTT-trancl-complete*:

$$(\text{gtt-lang } G)^+ \subseteq \text{gtt-lang } (GTT\text{-trancl } G)$$

*<proof>*

**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>*

**lemma**  *$\Delta$ -trancl-sound*:

**assumes**  $(p, q) \mid \in \mid \Delta\text{-trancl } A \ B$

**obtains**  $s \ t$  **where**  $(s, t) \in (\text{gtt-lang } (B, A))^+ \ p \mid \in \mid \text{gta-der } A \ s \ q \mid \in \mid \text{gta-der } B \ t$

*<proof>*

**lemma** *GTT-trancl-sound-aux*:

**assumes**  $p \mid \in \mid \text{gta-der } (TA \ (\text{rules } A) \ (\text{eps } A \ \mid \cup \mid (\Delta\text{-trancl } B \ A))) \ s$

**shows**  $\exists t. (s, t) \in (\text{gtt-lang } (A, B))^+ \ \wedge \ p \mid \in \mid \text{gta-der } A \ t$

*<proof>*

**lemma** *GTT-trancl-asound*:

$\text{agtt-lang } (GTT\text{-trancl } G) \subseteq \text{grancl-rel UNIV } (\text{agtt-lang } G)$

*<proof>*

**lemma** *GTT-trancl-sound*:

$\text{gtt-lang } (GTT\text{-trancl } G) \subseteq (\text{gtt-lang } G)^+$

*<proof>*

**lemma** *GTT-trancl-alang*:

$\text{agtt-lang } (GTT\text{-trancl } G) = \text{grancl-rel UNIV } (\text{agtt-lang } G)$

*<proof>*

**lemma** *GTT-trancl-lang*:

$\text{gtt-lang } (GTT\text{-trancl } G) = (\text{gtt-lang } G)^+$

*<proof>*

**end**

**theory** *Pair-Automaton*

**imports** *Tree-Automata-Complement GTT-Compose*

**begin**

## 4.9 Pair automaton and anchored GTTs

**definition** *pair-at-lang* ::  $( 'q, 'f) \text{ gtt} \Rightarrow ( 'q \times 'q) \text{ fset} \Rightarrow 'f \text{ gterm rel}$  **where**

$\text{pair-at-lang } \mathcal{G} \ Q = \{(s, t) \mid s \ t \ p \ q. \ q \mid \in \mid \text{gta-der } (\text{fst } \mathcal{G}) \ s \ \wedge \ p \mid \in \mid \text{gta-der } (\text{snd } \mathcal{G}) \ t \ \wedge \ (q, p) \mid \in \mid Q\}$

**lemma** *pair-at-lang-restr-states*:

$\text{pair-at-lang } \mathcal{G} \ Q = \text{pair-at-lang } \mathcal{G} \ (Q \ \mid \cap \mid (Q \ (\text{fst } \mathcal{G}) \ \mid \times \mid Q \ (\text{snd } \mathcal{G})))$

*<proof>*

**lemma** *pair-at-langE*:

**assumes**  $(s, t) \in \text{pair-at-lang } \mathcal{G} \ Q$

**obtains**  $q\ p$  **where**  $(q, p) \in Q$  **and**  $q \in \text{gta-der } (\text{fst } \mathcal{G})\ s$  **and**  $p \in \text{gta-der } (\text{snd } \mathcal{G})\ t$   
 ⟨proof⟩

**lemma** *pair-at-langI*:

**assumes**  $q \in \text{gta-der } (\text{fst } \mathcal{G})\ s$   $p \in \text{gta-der } (\text{snd } \mathcal{G})\ t$   $(q, p) \in Q$   
**shows**  $(s, t) \in \text{pair-at-lang } \mathcal{G}\ Q$   
 ⟨proof⟩

**lemma** *pair-at-lang-fun-states*:

**assumes** *finj-on*  $f$   $(Q\ (\text{fst } \mathcal{G}))$  **and** *finj-on*  $g$   $(Q\ (\text{snd } \mathcal{G}))$   
**and**  $Q \subseteq Q\ (\text{fst } \mathcal{G}) \times Q\ (\text{snd } \mathcal{G})$   
**shows**  $\text{pair-at-lang } \mathcal{G}\ Q = \text{pair-at-lang } (\text{map-prod } (\text{fmap-states-ta } f)\ (\text{fmap-states-ta } g)\ \mathcal{G})\ (\text{map-prod } f\ g\ |^{\cdot}\ Q)$   
 (**is**  $?LS = ?RS$ )  
 ⟨proof⟩

**lemma** *converse-pair-at-lang*:

$(\text{pair-at-lang } \mathcal{G}\ Q)^{-1} = \text{pair-at-lang } (\text{prod.swap } \mathcal{G})\ (Q|^{-1})$   
 ⟨proof⟩

**lemma** *pair-at-agtt*:

$\text{agtt-lang } \mathcal{G} = \text{pair-at-lang } \mathcal{G}\ (\text{fId-on } (\text{gtt-interface } \mathcal{G}))$   
 ⟨proof⟩

**definition**  $\Delta$ -*eps-pair* **where**

$\Delta\text{-eps-pair } \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2 \equiv Q_1 \mid O \mid \Delta_\varepsilon\ (\text{snd } \mathcal{G}_1)\ (\text{fst } \mathcal{G}_2) \mid O \mid Q_2$

**lemma** *pair-comp-sound1*:

**assumes**  $(s, t) \in \text{pair-at-lang } \mathcal{G}_1\ Q_1$   
**and**  $(t, u) \in \text{pair-at-lang } \mathcal{G}_2\ Q_2$   
**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⟩

**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**  $\exists t. (s, t) \in \text{pair-at-lang } \mathcal{G}_1\ Q_1 \wedge (t, u) \in \text{pair-at-lang } \mathcal{G}_2\ Q_2$   
 ⟨proof⟩

**lemma** *pair-comp-sound*:

$\text{pair-at-lang } \mathcal{G}_1\ Q_1\ O\ \text{pair-at-lang } \mathcal{G}_2\ Q_2 = \text{pair-at-lang } (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2)\ (\Delta\text{-eps-pair } \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2)$   
 ⟨proof⟩

**inductive-set**  $\Delta$ -*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**

*base* [*simp*]:  $(p, q) \in Q \Longrightarrow (p, q) \in \Delta\text{-Atrans-set } Q\ \mathcal{A}\ \mathcal{B}$   
 | *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**  $\Delta$ -Atrans-set-states:

$(p, q) \in \Delta\text{-Atrans-set } Q \mathcal{A} \mathcal{B} \implies (p, q) \in \text{fset } ((\text{fst } |\uparrow| Q \cup |\mathcal{Q} \mathcal{A}|) \times |\text{snd } |\uparrow| Q \cup |\mathcal{Q} \mathcal{B}|)$   
 $\langle \text{proof} \rangle$

**lemma** finite- $\Delta$ -Atrans-set: finite ( $\Delta$ -Atrans-set  $Q \mathcal{A} \mathcal{B}$ )

$\langle \text{proof} \rangle$

**context**

**includes** *fset.lifting*

**begin**

**lift-definition**  $\Delta\text{-Atrans} :: ('q \times 'q) \text{fset} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q \times 'q)$

**fset is**  $\Delta\text{-Atrans-set}$

$\langle \text{proof} \rangle$

**lemmas**  $\Delta\text{-Atrans-base}$  [*simp*] =  $\Delta\text{-Atrans-set.base}$  [*Transfer.transferred*]

**lemmas**  $\Delta\text{-Atrans-step}$  [*intro*] =  $\Delta\text{-Atrans-set.step}$  [*Transfer.transferred*]

**lemmas**  $\Delta\text{-Atrans-cases}$  =  $\Delta\text{-Atrans-set.cases}$  [*Transfer.transferred*]

**lemmas**  $\Delta\text{-Atrans-induct}$  [*consumes 1, case-names base step*] =  $\Delta\text{-Atrans-set.induct}$  [*Transfer.transferred*]

**end**

**abbreviation**  $\Delta\text{-Atrans-gtt } \mathcal{G} Q \equiv \Delta\text{-Atrans } Q (\text{fst } \mathcal{G}) (\text{snd } \mathcal{G})$

**lemma** *pair-trancl-sound1*:

**assumes**  $(s, t) \in (\text{pair-at-lang } \mathcal{G} Q)^+$

**shows**  $\exists q p. p \in |\text{gta-der } (\text{fst } \mathcal{G}) s \wedge q \in |\text{gta-der } (\text{snd } \mathcal{G}) t \wedge (p, q) \in |\Delta\text{-Atrans-gtt } \mathcal{G} Q$

$\langle \text{proof} \rangle$

**lemma** *pair-trancl-sound2*:

**assumes**  $(p, q) \in |\Delta\text{-Atrans-gtt } \mathcal{G} Q$

**and**  $p \in |\text{gta-der } (\text{fst } \mathcal{G}) s \wedge q \in |\text{gta-der } (\text{snd } \mathcal{G}) t$

**shows**  $(s, t) \in (\text{pair-at-lang } \mathcal{G} Q)^+ \langle \text{proof} \rangle$

**lemma** *pair-trancl-sound*:

$(\text{pair-at-lang } \mathcal{G} Q)^+ = \text{pair-at-lang } \mathcal{G} (\Delta\text{-Atrans-gtt } \mathcal{G} Q)$

$\langle \text{proof} \rangle$

**abbreviation** *fst-pair-cl*  $\mathcal{A} Q \equiv \text{TA } (\text{rules } \mathcal{A}) (\text{eps } \mathcal{A} \cup |\text{fId-on } (\mathcal{Q} \mathcal{A}) \cup |\mathcal{O}| Q))$

**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*  $\mathcal{G} Q = (\text{fst-pair-cl } (\text{fst } \mathcal{G}) Q, \text{TA } (\text{rules } (\text{snd } \mathcal{G})) (\text{eps } (\text{snd } \mathcal{G})))$

**lemma** *fst-pair-cl-eps*:

**assumes**  $(p, q) \in |(\text{eps } (\text{fst-pair-cl } \mathcal{A} Q))|^+$

**and**  $\mathcal{Q} \mathcal{A} \cup |\text{snd } |\uparrow| Q = \{|\}\}$

**shows**  $(p, q) \in |(\text{eps } \mathcal{A})|^+ \vee (\exists r. (p = r \vee (p, r) \in |(\text{eps } \mathcal{A})|^+) \wedge (r, q) \in |Q) \langle \text{proof} \rangle$

**lemma** *fst-pair-cl-res-aux*:

assumes  $Q \mathcal{A} \mid \cap \mid \text{snd} \mid \uparrow \mid Q = \{\mid\}$   
and  $q \mid \in \mid \text{ta-der} \mid (fst\text{-pair-cl } \mathcal{A} \ Q) \mid (term\text{-of-gterm } t)$   
shows  $\exists p. p \mid \in \mid \text{ta-der } \mathcal{A} \mid (term\text{-of-gterm } t) \wedge (q \mid \notin \mid Q \ \mathcal{A} \longrightarrow (p, q) \mid \in \mid Q) \wedge$   
 $(q \mid \in \mid Q \ \mathcal{A} \longrightarrow p = q) \langle proof \rangle$

**lemma** *restr-distjoing*:

assumes  $Q \mid \subseteq \mid Q \ \mathcal{A} \mid \times \mid Q \ \mathcal{B}$   
and  $Q \ \mathcal{A} \mid \cap \mid Q \ \mathcal{B} = \{\mid\}$   
shows  $Q \ \mathcal{A} \mid \cap \mid \text{snd} \mid \uparrow \mid Q = \{\mid\}$   
 $\langle proof \rangle$

**lemma** *pair-at-agtt-conv*:

assumes  $Q \mid \subseteq \mid Q \ (fst \ \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$  and  $Q \ (fst \ \mathcal{G}) \mid \cap \mid Q \ (\text{snd } \mathcal{G}) = \{\mid\}$   
shows *pair-at-lang*  $\mathcal{G} \ Q = \text{agtt-lang} \ (\text{pair-at-to-agtt } \mathcal{G} \ Q)$  (is ?LS = ?RS)  
 $\langle proof \rangle$

**definition** *pair-at-to-agtt' where*

*pair-at-to-agtt'*  $\mathcal{G} \ Q = (\text{let } \mathcal{A} = \text{fmap-states-ta } \text{Inl} \ (fst \ \mathcal{G}) \ \text{in}$   
 $\text{let } \mathcal{B} = \text{fmap-states-ta } \text{Inr} \ (\text{snd } \mathcal{G}) \ \text{in}$   
 $\text{let } Q' = Q \mid \cap \mid (Q \ (fst \ \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})) \ \text{in}$   
*pair-at-to-agtt*  $(\mathcal{A}, \mathcal{B}) \ (\text{map-prod } \text{Inl } \text{Inr} \mid \uparrow \mid Q')$ )

**lemma** *pair-at-agtt-cost*:

*pair-at-lang*  $\mathcal{G} \ Q = \text{agtt-lang} \ (\text{pair-at-to-agtt' } \mathcal{G} \ Q)$   
 $\langle proof \rangle$

**lemma**  $\Delta$ -*Atrans-states-stable*:

assumes  $Q \mid \subseteq \mid Q \ (fst \ \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$   
shows  $\Delta$ -*Atrans-gtt*  $\mathcal{G} \ Q \mid \subseteq \mid Q \ (fst \ \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$   
 $\langle proof \rangle$

**lemma**  $\Delta$ -*Atrans-map-prod*:

assumes *finj-on*  $f \ (Q \ (fst \ \mathcal{G}))$  and *finj-on*  $g \ (Q \ (\text{snd } \mathcal{G}))$   
and  $Q \mid \subseteq \mid Q \ (fst \ \mathcal{G}) \mid \times \mid Q \ (\text{snd } \mathcal{G})$   
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)  
 $\langle proof \rangle$

**definition** *Q-pow where*

*Q-pow*  $Q \ \mathcal{S}_1 \ \mathcal{S}_2 =$   
 $\{\mid (\text{Wrapp } X, \text{Wrapp } Y) \mid X \ Y \ p \ q. X \mid \in \mid fPow \ \mathcal{S}_1 \wedge Y \mid \in \mid fPow \ \mathcal{S}_2 \wedge p \mid \in \mid X$   
 $\wedge q \mid \in \mid Y \wedge (p, q) \mid \in \mid Q \mid \}$

**lemma** *Q-pow-fmember*:

$(X, Y) \mid \in \mid Q\text{-pow } Q \ \mathcal{S}_1 \ \mathcal{S}_2 \longleftrightarrow (\exists p \ q. \text{ex } X \mid \in \mid fPow \ \mathcal{S}_1 \wedge \text{ex } Y \mid \in \mid fPow \ \mathcal{S}_2$   
 $\wedge p \mid \in \mid \text{ex } X \wedge q \mid \in \mid \text{ex } Y \wedge (p, q) \mid \in \mid Q)$   
 $\langle proof \rangle$

**lemma** *pair-automaton-det-lang-sound-complete:*

*pair-at-lang*  $\mathcal{G} \ Q = \text{pair-at-lang} (\text{map-both } ps\text{-ta } \mathcal{G}) (\mathcal{Q}\text{-pow } Q (\mathcal{Q} (\text{fst } \mathcal{G})) (\mathcal{Q} (\text{snd } \mathcal{G})))$  (is ?LS = ?RS)  
 ⟨proof⟩

**lemma** *pair-automaton-complement-sound-complete:*

**assumes** *partially-completely-defined-on*  $\mathcal{A} \ \mathcal{F}$  **and** *partially-completely-defined-on*  $\mathcal{B} \ \mathcal{F}$

**and** *ta-det*  $\mathcal{A}$  **and** *ta-det*  $\mathcal{B}$

**shows** *pair-at-lang*  $(\mathcal{A}, \mathcal{B}) (\mathcal{Q} \ \mathcal{A} \ |\times| \ \mathcal{Q} \ \mathcal{B} \ |-| \ Q) = \text{gterms} (\text{fset } \mathcal{F}) \times \text{gterms} (\text{fset } \mathcal{F}) - \text{pair-at-lang} (\mathcal{A}, \mathcal{B}) \ Q$

⟨proof⟩

**end**

**theory** *AGTT*

**imports** *GTT GTT-Transitive-Closure Pair-Automaton*

**begin**

**definition** *AGTT-union where*

*AGTT-union*  $\mathcal{G}_1 \ \mathcal{G}_2 \equiv (\text{ta-union} (\text{fst } \mathcal{G}_1) (\text{fst } \mathcal{G}_2),$   
 $\text{ta-union} (\text{snd } \mathcal{G}_1) (\text{snd } \mathcal{G}_2))$

**abbreviation** *AGTT-union' where*

*AGTT-union'*  $\mathcal{G}_1 \ \mathcal{G}_2 \equiv \text{AGTT-union} (\text{fmap-states-gtt Inl } \mathcal{G}_1) (\text{fmap-states-gtt Inr } \mathcal{G}_2)$

**lemma** *disj-gtt-states-disj-fst-ta-states:*

**assumes** *dist-st: gtt-states*  $\mathcal{G}_1 \ |\cap| \ \text{gtt-states } \mathcal{G}_2 = \{\|\}$

**shows**  $\mathcal{Q} (\text{fst } \mathcal{G}_1) \ |\cap| \ \mathcal{Q} (\text{fst } \mathcal{G}_2) = \{\|\}$

⟨proof⟩

**lemma** *disj-gtt-states-disj-snd-ta-states:*

**assumes** *dist-st: gtt-states*  $\mathcal{G}_1 \ |\cap| \ \text{gtt-states } \mathcal{G}_2 = \{\|\}$

**shows**  $\mathcal{Q} (\text{snd } \mathcal{G}_1) \ |\cap| \ \mathcal{Q} (\text{snd } \mathcal{G}_2) = \{\|\}$

⟨proof⟩

**lemma** *ta-der-not-contains-undefined-state:*

**assumes**  $q \notin \mathcal{Q} \ T$  **and** *ground*  $t$

**shows**  $q \notin \text{ta-der } T \ t$

⟨proof⟩

**lemma** *AGTT-union-sound1:*

**assumes** *dist-st: gtt-states*  $\mathcal{G}_1 \ |\cap| \ \text{gtt-states } \mathcal{G}_2 = \{\|\}$

**shows** *agtt-lang*  $(\text{AGTT-union } \mathcal{G}_1 \ \mathcal{G}_2) \subseteq \text{agtt-lang } \mathcal{G}_1 \cup \text{agtt-lang } \mathcal{G}_2$

⟨proof⟩

**lemma** *AGTT-union-sound2:*

**shows**  $\text{agtt-lang } \mathcal{G}_1 \subseteq \text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2)$   
 $\text{agtt-lang } \mathcal{G}_2 \subseteq \text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2)$   
 $\langle \text{proof} \rangle$

**lemma** *AGTT-union-sound*:

**assumes**  $\text{dist-st: gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$   
**shows**  $\text{agtt-lang } (\text{AGTT-union } \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cup \text{agtt-lang } \mathcal{G}_2$   
 $\langle \text{proof} \rangle$

**lemma** *AGTT-union'-sound*:

**fixes**  $\mathcal{G}_1 :: ('q, 'f) \text{ gtt}$  **and**  $\mathcal{G}_2 :: ('q, 'f) \text{ gtt}$   
**shows**  $\text{agtt-lang } (\text{AGTT-union}' \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cup \text{agtt-lang } \mathcal{G}_2$   
 $\langle \text{proof} \rangle$

#### 4.10 Ancho rd gtt compositon

**definition** *AGTT-comp* ::  $('q, 'f) \text{ gtt} \Rightarrow ('q, 'f) \text{ gtt} \Rightarrow ('q, 'f) \text{ gtt}$  **where**

$\text{AGTT-comp } \mathcal{G}_1 \mathcal{G}_2 = (\text{let } (\mathcal{A}, \mathcal{B}) = (\text{fst } \mathcal{G}_1, \text{snd } \mathcal{G}_2) \text{ in}$   
 $(\text{TA } (\text{rules } \mathcal{A}) (\text{eps } \mathcal{A} \mid \cup \mid (\Delta_\varepsilon (\text{snd } \mathcal{G}_1) (\text{fst } \mathcal{G}_2) \mid \cap \mid (\text{gtt-interface } \mathcal{G}_1 \mid \times \mid$   
 $\text{gtt-interface } \mathcal{G}_2))))$ ,  
 $\text{TA } (\text{rules } \mathcal{B}) (\text{eps } \mathcal{B}))$ )

**abbreviation** *AGTT-comp'* **where**

$\text{AGTT-comp}' \mathcal{G}_1 \mathcal{G}_2 \equiv \text{AGTT-comp } (\text{fmap-states-gtt Inl } \mathcal{G}_1) (\text{fmap-states-gtt Inr } \mathcal{G}_2)$

**lemma** *AGTT-comp-sound*:

**assumes**  $\text{gtt-states } \mathcal{G}_1 \mid \cap \mid \text{gtt-states } \mathcal{G}_2 = \{\mid\}$   
**shows**  $\text{agtt-lang } (\text{AGTT-comp } \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \text{ O } \text{agtt-lang } \mathcal{G}_2$   
 $\langle \text{proof} \rangle$

**lemma** *AGTT-comp'-sound*:

$\text{agtt-lang } (\text{AGTT-comp}' \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \text{ O } \text{agtt-lang } \mathcal{G}_2$   
 $\langle \text{proof} \rangle$

#### 4.11 Ancho rd gtt transitivity

**definition** *AGTT-trancl* ::  $('q, 'f) \text{ gtt} \Rightarrow ('q + 'q, 'f) \text{ gtt}$  **where**

$\text{AGTT-trancl } \mathcal{G} = (\text{let } \mathcal{A} = \text{fmap-states-ta Inl } (\text{fst } \mathcal{G}) \text{ in}$   
 $(\text{TA } (\text{rules } \mathcal{A}) (\text{eps } \mathcal{A} \mid \cup \mid \text{map-prod CInl CInr } \mid \uparrow \mid (\Delta\text{-Atrans-gtt } \mathcal{G} (\text{fId-on}$   
 $\text{gtt-interface } \mathcal{G}))))$ ,  
 $\text{TA } (\text{map-ta-rule CInr id } \mid \uparrow \mid (\text{rules } (\text{snd } \mathcal{G})) (\text{map-both CInr } \mid \uparrow \mid (\text{eps } (\text{snd } \mathcal{G}))))$ )

**lemma** *AGTT-trancl-sound*:

**shows**  $\text{agtt-lang } (\text{AGTT-trancl } \mathcal{G}) = (\text{agtt-lang } \mathcal{G})^+$   
 $\langle \text{proof} \rangle$

#### 4.12 Ancho rd gtt intersection

**definition** *AGTT-inter* **where**

$$\begin{aligned} AGTT\text{-inter } \mathcal{G}_1 \mathcal{G}_2 &\equiv (\text{prod-ta } (\text{fst } \mathcal{G}_1) (\text{fst } \mathcal{G}_2), \\ &\quad \text{prod-ta } (\text{snd } \mathcal{G}_1) (\text{snd } \mathcal{G}_2)) \end{aligned}$$

**lemma** *AGTT-inter-sound*:

$$\text{agtt-lang } (AGTT\text{-inter } \mathcal{G}_1 \mathcal{G}_2) = \text{agtt-lang } \mathcal{G}_1 \cap \text{agtt-lang } \mathcal{G}_2 \text{ (is ?Ls = ?Rs)}$$

*<proof>*

### 4.13 Anchor 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>*

**lemma** *agtt-only-reach-lang*:

$$\text{agtt-lang } (\text{gtt-only-reach } \mathcal{G}) = \text{agtt-lang } \mathcal{G}$$

*<proof>*

**lemma** *trim-agtt-lang [simp]*:

$$\text{agtt-lang } (\text{trim-agtt } G) = \text{agtt-lang } G$$

*<proof>*

**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* ::  $'f \text{ gterm} \Rightarrow 'g \text{ gterm} \Rightarrow ('f \text{ option} \times 'g \text{ option}) \text{ gterm}$  **where**  
 $\text{gpair } s \ t = \text{glabel } (\lambda p. (\text{gfun-at } s \ p, \text{gfun-at } t \ p)) (\text{gunion } (\text{gdomain } s) (\text{gdomain } t))$

We provide an efficient implementation of *gpair*.

**definition** *zip-fill* ::  $'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \text{ option} \times 'b \text{ option}) \text{ list}$  **where**  
 $\text{zip-fill } xs \ ys = \text{zip } (\text{map } \text{Some } xs \ @ \ \text{replicate } (\text{length } ys - \text{length } xs) \ \text{None})$   
 $\quad (\text{map } \text{Some } ys \ @ \ \text{replicate } (\text{length } xs - \text{length } ys) \ \text{None})$

**lemma** *zip-fill-code [code]*:

$$\begin{aligned} \text{zip-fill } xs \ [] &= \text{map } (\lambda x. (\text{Some } x, \text{None})) \ xs \\ \text{zip-fill } [] \ ys &= \text{map } (\lambda y. (\text{None}, \text{Some } y)) \ ys \end{aligned}$$

```
zip-fill (x # xs) (y # ys) = (Some x, Some y) # zip-fill xs ys
⟨proof⟩
```

```
lemma length-zip-fill [simp]:
  length (zip-fill xs ys) = max (length xs) (length ys)
⟨proof⟩
```

```
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)
⟨proof⟩
```

```
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⟩
```

```
lemma gpair-code [code]:
  gpair s t = gpair-impl (Some s) (Some t)
⟨proof⟩
```

```
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
⟨proof⟩
```

```
lemma gpair-assoc:
  defines f ≡ λ(f, gh). (f, gh ≫= fst, gh ≫= snd)
  defines g ≡ λ(fg, h). (fg ≫= fst, fg ≫= snd, h)
  shows map-gterm f (gpair s (gpair t u)) = map-gterm g (gpair (gpair s t) u)
⟨proof⟩
```

## 5.2 Decoding of pairs

**fun** *gcollapse* :: 'f option gterm ⇒ 'f gterm option **where**  
*gcollapse* (GFun None \_) = None  
| *gcollapse* (GFun (Some f) ts) = Some (GFun f (map the (filter (λt. ¬ Option.is-none t) (map *gcollapse* ts))))

**lemma** *gcollapse-groot-None* [*simp*]:  
*groot-sym* t = None ⇒ *gcollapse* t = None  
*fst* (groot t) = None ⇒ *gcollapse* t = None  
⟨*proof*⟩

**definition** *gfst* :: ('f option × 'g option) gterm ⇒ 'f gterm **where**  
*gfst* = the ∘ *gcollapse* ∘ map-gterm *fst*

**definition** *gsnd* :: ('f option × 'g option) gterm ⇒ 'g gterm **where**  
*gsnd* = the ∘ *gcollapse* ∘ map-gterm *snd*

**lemma** *filter-less-upt*:  
*[i←[i..<m] . i < n] = [i..<min n m]*  
⟨*proof*⟩

**lemma** *gcollapse-aux*:  
**assumes** *gposs* s = {p. p ∈ *gposs* t ∧ *gfun-at* t p ≠ Some None}  
**shows** *gposs* (the (*gcollapse* t)) = *gposs* s  
∧ p. p ∈ *gposs* s ⇒ *gfun-at* (the (*gcollapse* t)) p = (*gfun-at* t p ≫ id)  
⟨*proof*⟩

**lemma** *gfst-gpair*:  
*gfst* (gpair s t) = s  
⟨*proof*⟩

**lemma** *gsnd-gpair*:  
*gsnd* (gpair s t) = t  
⟨*proof*⟩

**lemma** *gpair-impl-None-Inv*:  
map-gterm (the ∘ *snd*) (gpair-impl None (Some t)) = t  
⟨*proof*⟩

## 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))  
⟨*proof*⟩

**lemma** *gpair-context2*:  
**assumes** ∧ i. i < *length* ts ⇒ ts ! i = *gpair* (ss ! i) (us ! i)

**and**  $\text{length } ss = \text{length } ts$  **and**  $\text{length } us = \text{length } ts$   
**shows**  $\text{GFun } (Some\ f, Some\ h)\ ts = \text{gpair } (\text{GFun } f\ ss)\ (\text{GFun } h\ us)$   
 $\langle \text{proof} \rangle$

**lemma** *map-funs-term-some-gpair*:  
**shows**  $\text{gpair } t\ t = \text{map-gterm } (\lambda f. (Some\ f, Some\ f))\ t$   
 $\langle \text{proof} \rangle$

**lemma** *gpair-inject [simp]*:  
 $\text{gpair } s\ t = \text{gpair } s'\ t' \longleftrightarrow s = s' \wedge t = t'$   
 $\langle \text{proof} \rangle$

**abbreviation** *gterm-to-None-Some*  $:: 'f\ \text{gterm} \Rightarrow ('f\ \text{option} \times 'f\ \text{option})\ \text{gterm}$   
**where**

$\text{gterm-to-None-Some } t \equiv \text{map-gterm } (\lambda f. (None, Some\ f))\ t$

**abbreviation** *gterm-to-Some-None*  $t \equiv \text{map-gterm } (\lambda f. (Some\ f, None))\ t$

**lemma** *inj-gterm-to-None-Some*:  $\text{inj } \text{gterm-to-None-Some}$   
 $\langle \text{proof} \rangle$

**lemma** *zip-fill1*:  
**assumes**  $\text{length } ss < \text{length } ts$   
**shows**  $\text{zip-fill } ss\ ts = \text{zip } (\text{map } Some\ ss)\ (\text{map } Some\ (\text{take } (\text{length } ss)\ ts))\ @$   
 $\text{map } (\lambda x. (None, Some\ x))\ (\text{drop } (\text{length } ss)\ ts)$   
 $\langle \text{proof} \rangle$

**lemma** *zip-fill2*:  
**assumes**  $\text{length } ts < \text{length } ss$   
**shows**  $\text{zip-fill } ss\ ts = \text{zip } (\text{map } Some\ (\text{take } (\text{length } ts)\ ss))\ (\text{map } Some\ ts)\ @$   
 $\text{map } (\lambda x. (Some\ x, None))\ (\text{drop } (\text{length } ts)\ ss)$   
 $\langle \text{proof} \rangle$

**lemma** *not-gposs-append [simp]*:  
**assumes**  $p \notin \text{gposs } t$   
**shows**  $p\ @\ q \in \text{gposs } t = \text{False}$   $\langle \text{proof} \rangle$

**lemma** *gfun-at-gpair*:  
 $\text{gfun-at } (\text{gpair } s\ t)\ p = (\text{if } p \in \text{gposs } s\ \text{then } (\text{if } p \in \text{gposs } t$   
 $\text{then } Some\ (\text{gfun-at } s\ p, \text{gfun-at } t\ p)$   
 $\text{else } Some\ (\text{gfun-at } s\ p, None))\ \text{else}$   
 $(\text{if } p \in \text{gposs } t\ \text{then } Some\ (None, \text{gfun-at } t\ p)\ \text{else } None))$   
 $\langle \text{proof} \rangle$

**lemma** *gposs-of-gpair* [simp]:

**shows**  $gposs (gpair\ s\ t) = gposs\ s \cup gposs\ t$   
*<proof>*

**lemma** *poss-to-gpair-poss*:

$p \in gposs\ s \implies p \in gposs (gpair\ s\ t)$   
 $p \in gposs\ t \implies p \in gposs (gpair\ s\ t)$   
*<proof>*

**lemma** *gsubt-at-gpair-poss*:

**assumes**  $p \in gposs\ s$  **and**  $p \in gposs\ t$   
**shows**  $gsubt-at (gpair\ s\ t)\ p = gpair (gsubt-at\ s\ p) (gsubt-at\ t\ p)$  *<proof>*

**lemma** *subst-at-gpair-nt-poss-Some-None*:

**assumes**  $p \in gposs\ s$  **and**  $p \notin gposs\ t$   
**shows**  $gsubt-at (gpair\ s\ t)\ p = gterm-to-Some-None (gsubt-at\ s\ p)$  *<proof>*

**lemma** *subst-at-gpair-nt-poss-None-Some*:

**assumes**  $p \in gposs\ t$  **and**  $p \notin gposs\ s$   
**shows**  $gsubt-at (gpair\ s\ t)\ p = gterm-to-None-Some (gsubt-at\ t\ p)$  *<proof>*

**lemma** *gpair-ctxt-decomposition*:

**fixes**  $C$  **defines**  $p \equiv ghole-pos\ C$   
**assumes**  $p \notin gposs\ s$  **and**  $gpair\ s\ t = C \langle gterm-to-None-Some\ u \rangle_G$   
**shows**  $gpair\ s (gctxt-at-pos\ t\ p) \langle v \rangle_G = C \langle gterm-to-None-Some\ v \rangle_G$   
*<proof>*

**lemma** *groot-gpair* [simp]:

$fst (groot (gpair\ s\ t)) = (Some (fst (groot\ s)), Some (fst (groot\ t)))$   
*<proof>*

**lemma** *ground-ctxt-adapt-ground* [intro]:

**assumes**  $ground-ctxt\ C$   
**shows**  $ground-ctxt (adapt-vars-ctxt\ C)$   
*<proof>*

**lemma** *adapt-vars-ctxt2* :

**assumes**  $ground-ctxt\ C$   
**shows**  $adapt-vars-ctxt (adapt-vars-ctxt\ C) = adapt-vars-ctxt\ C$  *<proof>*

## 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**  $i < \text{length } ts$   
**shows**  $\text{gdecode-nth } (\text{gencode } ts) \ i = ts \ ! \ i$   
 $\langle \text{proof} \rangle$

**definition** *gdecode* :: '*f option list gterm*  $\Rightarrow$  '*f gterm list* **where**  
 $\text{gdecode } t = (\text{case } t \text{ of } \text{GFun } f \ ts \Rightarrow \text{map } (\lambda i. \text{gdecode-nth } t \ i) \ [0..<\text{length } f])$

**lemma** *gdecode-gencode*:  
 $\text{gdecode } (\text{gencode } ts) = ts$   
 $\langle \text{proof} \rangle$

**definition** *gencode-impl* :: '*f gterm option list*  $\Rightarrow$  '*f option list gterm* **where**  
 $\text{gencode-impl } ts = \text{glabel } (\lambda p. \text{map } (\lambda t. t \ggg (\lambda t. \text{gfun-at } t \ p)) \ ts) \ (\text{g unions } (\text{map } (\text{case-option } (\text{GFun } () \ [])) \ \text{gdomain}) \ ts)$

**lemma** *gencode-code* [*code*]:  
 $\text{gencode } ts = \text{gencode-impl } (\text{map } \text{Some } ts)$   
 $\langle \text{proof} \rangle$

**lemma** *gencode-singleton*:  
 $\text{gencode } [t] = \text{map-gterm } (\lambda f. [\text{Some } f]) \ t$   
 $\langle \text{proof} \rangle$

**lemma** *gencode-pair*:  
 $\text{gencode } [t, u] = \text{map-gterm } (\lambda (f, g). [f, g]) \ (\text{gpair } t \ u)$   
 $\langle \text{proof} \rangle$

## 5.5 RRn relations

**definition** *RR1-spec* **where**  
 $\text{RR1-spec } A \ T \longleftrightarrow \mathcal{L} \ A = T$

**definition** *RR2-spec* **where**  
 $\text{RR2-spec } A \ T \longleftrightarrow \mathcal{L} \ A = \{\text{gpair } t \ u \mid t \ u. (t, u) \in T\}$

**definition** *RRn-spec* **where**  
 $\text{RRn-spec } n \ A \ R \longleftrightarrow \mathcal{L} \ A = \text{gencode } \text{' } R \wedge (\forall ts \in R. \text{length } ts = n)$

**lemma** *RR1-to-RRn-spec*:  
**assumes** *RR1-spec*  $A \ T$   
**shows** *RRn-spec*  $1 \ (\text{fmap-funs-reg } (\lambda f. [\text{Some } f]) \ A) \ ((\lambda t. [t]) \text{' } T)$   
 $\langle \text{proof} \rangle$

**lemma** *RR2-to-RRn-spec*:  
**assumes** *RR2-spec*  $A \ T$   
**shows** *RRn-spec*  $2 \ (\text{fmap-funs-reg } (\lambda (f, g). [f, g]) \ A) \ ((\lambda (t, u). [t, u]) \text{' } T)$   
 $\langle \text{proof} \rangle$

**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>*

**lemma** *relabel-RR1-spec* [*simp*]:  
*RR1-spec* (*relabel-reg* *A*) *T*  $\longleftrightarrow$  *RR1-spec* *A T*  
*<proof>*

**lemma** *relabel-RR2-spec* [*simp*]:  
*RR2-spec* (*relabel-reg* *A*) *T*  $\longleftrightarrow$  *RR2-spec* *A T*  
*<proof>*

**lemma** *relabel-RRn-spec* [*simp*]:  
*RRn-spec* *n* (*relabel-reg* *A*) *T*  $\longleftrightarrow$  *RRn-spec* *n A T*  
*<proof>*

**lemma** *trim-RR1-spec* [*simp*]:  
*RR1-spec* (*trim-reg* *A*) *T*  $\longleftrightarrow$  *RR1-spec* *A T*  
*<proof>*

**lemma** *trim-RR2-spec* [*simp*]:  
*RR2-spec* (*trim-reg* *A*) *T*  $\longleftrightarrow$  *RR2-spec* *A T*  
*<proof>*

**lemma** *trim-RRn-spec* [*simp*]:  
*RRn-spec* *n* (*trim-reg* *A*) *T*  $\longleftrightarrow$  *RRn-spec* *n A T*  
*<proof>*

**lemma** *swap-RR2-spec*:  
**assumes** *RR2-spec* *A R*  
**shows** *RR2-spec* (*fmap-funs-reg* *prod.swap* *A*) (*prod.swap* ‘ *R*) *<proof>*

## 5.6 Nullary automata

**lemma** *false-RRn-spec*:  
*RRn-spec* *n* *empty-reg* {}  
*<proof>*

**lemma** *true-RR0-spec*:  
*RRn-spec* 0 (*Reg* {|*q*|} (*TA* {| $\square$   $\square$   $\rightarrow$  *q*|} {| $\square$ |})) {| $\square$ |}  
*<proof>*

## 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**  $\text{pair-eps } A B \equiv (\lambda (p, q). ((\text{Some } (\text{fst } p), q), (\text{Some } (\text{snd } p), q))) \mid \uparrow$   
 $(\text{eps } A \mid \times \mid \text{finsert } \text{None } (\text{Some } \mid \uparrow \mid \mathcal{Q} B))$

**abbreviation**  $\text{pair-rule} \equiv (\lambda (ra, rb). \text{TA-rule } (\text{Some } (r\text{-root } ra), \text{Some } (r\text{-root } rb))$   
 $(\text{zip-fill } (r\text{-lhs-states } ra) (r\text{-lhs-states } rb)) (\text{Some } (r\text{-rhs } ra), \text{Some } (r\text{-rhs } rb)))$

**lemma**  $\text{lift-Some-None-pord-swap}$  [simp]:

$\text{prod.swap} \circ \text{lift-Some-None} = \text{lift-None-Some}$

$\text{prod.swap} \circ \text{lift-None-Some} = \text{lift-Some-None}$

$\langle \text{proof} \rangle$

**lemma**  $\text{eps-to-pair-eps-Some-None}$ :

$(p, q) \mid \in \mid \text{eps } \mathcal{A} \implies (\text{lift-Some-None } p, \text{lift-Some-None } q) \mid \in \mid \text{pair-eps } \mathcal{A} \mathcal{B}$

$\langle \text{proof} \rangle$

**definition**  $\text{pair-automaton} :: ('p, 'f) \text{ta} \Rightarrow ('q, 'g) \text{ta} \Rightarrow ('p \text{ option} \times 'q \text{ option}, 'f$   
 $\text{option} \times 'g \text{ option}) \text{ta}$  **where**

$\text{pair-automaton } A B = \text{TA}$

$(\text{map-ta-rule } \text{lift-Some-None } \text{lift-Some-None} \mid \uparrow \mid \text{rules } A \mid \cup \mid$

$\text{map-ta-rule } \text{lift-None-Some } \text{lift-None-Some} \mid \uparrow \mid \text{rules } B \mid \cup \mid$

$\text{pair-rule} \mid \uparrow \mid (\text{rules } A \mid \times \mid \text{rules } B))$

$(\text{pair-eps } A B \mid \cup \mid \text{map-both } \text{prod.swap} \mid \uparrow \mid (\text{pair-eps } B A))$

**definition**  $\text{pair-automaton-reg}$  **where**

$\text{pair-automaton-reg } R L = \text{Reg } (\text{Some } \mid \uparrow \mid \text{fin } R \mid \times \mid \text{Some } \mid \uparrow \mid \text{fin } L) (\text{pair-automaton}$   
 $(\text{ta } R) (\text{ta } L))$

**lemma**  $\text{pair-automaton-eps-simps}$ :

$(\text{lift-Some-None } p, p') \mid \in \mid \text{eps } (\text{pair-automaton } A B) \longleftrightarrow (\text{lift-Some-None } p, p')$   
 $\mid \in \mid \text{pair-eps } A B$

$(q, \text{lift-Some-None } q') \mid \in \mid \text{eps } (\text{pair-automaton } A B) \longleftrightarrow (q, \text{lift-Some-None}$   
 $q') \mid \in \mid \text{pair-eps } A B$

$\langle \text{proof} \rangle$

**lemma**  $\text{pair-automaton-eps-Some-SomeD}$ :

$((\text{Some } p, \text{Some } p'), r) \mid \in \mid \text{eps } (\text{pair-automaton } A B) \implies \text{fst } r \neq \text{None} \wedge \text{snd } r$   
 $\neq \text{None} \wedge (\text{Some } p = \text{fst } r \vee \text{Some } p' = \text{snd } r) \wedge$

$(\text{Some } p \neq \text{fst } r \longrightarrow (p, \text{the } (\text{fst } r)) \mid \in \mid (\text{eps } A)) \wedge (\text{Some } p' \neq \text{snd } r \longrightarrow (p',$   
 $\text{the } (\text{snd } r)) \mid \in \mid (\text{eps } B))$

$\langle \text{proof} \rangle$

**lemma**  $\text{pair-automaton-eps-Some-SomeD2}$ :

$(r, (\text{Some } p, \text{Some } p')) \mid \in \mid \text{eps } (\text{pair-automaton } A B) \implies \text{fst } r \neq \text{None} \wedge \text{snd } r$   
 $\neq \text{None} \wedge (\text{fst } r = \text{Some } p \vee \text{snd } r = \text{Some } p') \wedge$

$(\text{fst } r \neq \text{Some } p \longrightarrow (\text{the } (\text{fst } r), p) \mid \in \mid (\text{eps } A)) \wedge (\text{snd } r \neq \text{Some } p' \longrightarrow (\text{the}$   
 $(\text{snd } r), p') \mid \in \mid (\text{eps } B))$

$\langle \text{proof} \rangle$

**lemma**  $\text{pair-eps-Some-None}$ :

**fixes**  $p\ q\ q'$   
**defines**  $l \equiv (p, q)$  **and**  $r \equiv \text{lift-Some-None } q'$   
**assumes**  $(l, r) \mid\in\mid (\text{eps } (\text{pair-automaton } A\ B)) \mid^+ \mid$   
**shows**  $q = \text{None} \wedge p \neq \text{None} \wedge (\text{the } p, q') \mid\in\mid (\text{eps } A) \mid^+ \mid \langle \text{proof} \rangle$

**lemma** *pair-eps-Some-Some*:

**fixes**  $p\ q$   
**defines**  $l \equiv (\text{Some } p, \text{Some } q)$   
**assumes**  $(l, r) \mid\in\mid (\text{eps } (\text{pair-automaton } A\ B)) \mid^+ \mid$   
**shows**  $\text{fst } r \neq \text{None} \wedge \text{snd } r \neq \text{None} \wedge$   
 $(\text{fst } l \neq \text{fst } r \longrightarrow (p, \text{the } (\text{fst } r)) \mid\in\mid (\text{eps } A) \mid^+ \mid) \wedge$   
 $(\text{snd } l \neq \text{snd } r \longrightarrow (q, \text{the } (\text{snd } r)) \mid\in\mid (\text{eps } B) \mid^+ \mid)$   
 $\langle \text{proof} \rangle$

**lemma** *pair-eps-Some-Some2*:

**fixes**  $p\ q$   
**defines**  $r \equiv (\text{Some } p, \text{Some } q)$   
**assumes**  $(l, r) \mid\in\mid (\text{eps } (\text{pair-automaton } A\ B)) \mid^+ \mid$   
**shows**  $\text{fst } l \neq \text{None} \wedge \text{snd } l \neq \text{None} \wedge$   
 $(\text{fst } l \neq \text{fst } r \longrightarrow (\text{the } (\text{fst } l), p) \mid\in\mid (\text{eps } A) \mid^+ \mid) \wedge$   
 $(\text{snd } l \neq \text{snd } r \longrightarrow (\text{the } (\text{snd } l), q) \mid\in\mid (\text{eps } B) \mid^+ \mid)$   
 $\langle \text{proof} \rangle$

**lemma** *map-pair-automaton*:

$\text{pair-automaton } (\text{fmap-funs-ta } f\ A) (\text{fmap-funs-ta } g\ B) =$   
 $\text{fmap-funs-ta } (\lambda(a, b). (\text{map-option } f\ a, \text{map-option } g\ b)) (\text{pair-automaton } A\ B)$   
**(is ?Ls = ?Rs)**  
 $\langle \text{proof} \rangle$

**lemmas** *map-pair-automaton-12* =

$\text{map-pair-automaton}[\text{of } - - \text{id}, \text{unfolded } \text{fmap-funs-ta-id } \text{option.map-id}]$   
 $\text{map-pair-automaton}[\text{of } \text{id} - -, \text{unfolded } \text{fmap-funs-ta-id } \text{option.map-id}]$

**lemma** *fmap-states-funs-ta-commute*:

$\text{fmap-states-ta } f (\text{fmap-funs-ta } g\ A) = \text{fmap-funs-ta } g (\text{fmap-states-ta } f\ A)$   
 $\langle \text{proof} \rangle$

**lemma** *states-pair-automaton*:

$\mathcal{Q} (\text{pair-automaton } A\ B) \mid\subseteq\mid (\text{finsert } \text{None } (\text{Some } \mid\uparrow\mid \mathcal{Q}\ A) \mid\times\mid (\text{finsert } \text{None}$   
 $(\text{Some } \mid\uparrow\mid \mathcal{Q}\ B)))$   
 $\langle \text{proof} \rangle$

**lemma** *swap-pair-automaton*:

**assumes**  $(p, q) \mid\in\mid \text{ta-der } (\text{pair-automaton } A\ B) (\text{term-of-gterm } t)$   
**shows**  $(q, p) \mid\in\mid \text{ta-der } (\text{pair-automaton } B\ A) (\text{term-of-gterm } (\text{map-gterm } \text{prod.swap}$   
 $t))$   
 $\langle \text{proof} \rangle$

**lemma** *to-ta-der-pair-automaton*:

$p \mid\in\mid ta\text{-der } A \text{ (term-of-gterm } t) \implies$   
 $(Some\ p, None) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm (map-gterm$   
 $(\lambda f. (Some\ f, None))\ t))$   
 $q \mid\in\mid ta\text{-der } B \text{ (term-of-gterm } u) \implies$   
 $(None, Some\ q) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm (map-gterm$   
 $(\lambda f. (None, Some\ f))\ u))$   
 $p \mid\in\mid ta\text{-der } A \text{ (term-of-gterm } t) \implies q \mid\in\mid ta\text{-der } B \text{ (term-of-gterm } u) \implies$   
 $(Some\ p, Some\ q) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm (gpair } t\ u))$   
 $\langle proof \rangle$

**lemma** *from-ta-der-pair-automaton*:

$(None, None) \mid\notin\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm } s)$   
 $(Some\ p, None) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm } s) \implies$   
 $\exists t. p \mid\in\mid ta\text{-der } A \text{ (term-of-gterm } t) \wedge s = \text{map-gterm } (\lambda f. (Some\ f, None))\ t$   
 $(None, Some\ q) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm } s) \implies$   
 $\exists u. q \mid\in\mid ta\text{-der } B \text{ (term-of-gterm } u) \wedge s = \text{map-gterm } (\lambda f. (None, Some\ f))\ u$   
 $(Some\ p, Some\ q) \mid\in\mid ta\text{-der (pair-automaton } A\ B) \text{ (term-of-gterm } s) \implies$   
 $\exists t\ u. p \mid\in\mid ta\text{-der } A \text{ (term-of-gterm } t) \wedge q \mid\in\mid ta\text{-der } B \text{ (term-of-gterm } u) \wedge s =$   
 $gpair\ t\ u$   
 $\langle proof \rangle$

**lemma** *diagonal-automaton*:

**assumes**  $RR1\text{-spec } A\ R$   
**shows**  $RR2\text{-spec (fmap-funs-reg } (\lambda f. (Some\ f, Some\ f))\ A) \{(s, s) \mid s. s \in R\}$   
 $\langle proof \rangle$

**lemma** *pair-automaton*:

**assumes**  $RR1\text{-spec } A\ T\ RR1\text{-spec } B\ U$   
**shows**  $RR2\text{-spec (pair-automaton-reg } A\ B) (T \times U)$   
 $\langle proof \rangle$

**lemma** *pair-automaton'*:

**shows**  $\mathcal{L} \text{ (pair-automaton-reg } A\ B) = \text{case-prod } gpair\ ' (\mathcal{L}\ A \times \mathcal{L}\ B)$   
 $\langle proof \rangle$

## 5.8 Collapsing

cf. *gcollapse*.

**fun** *collapse-state-list where*

$collapse\text{-state-list } Qn\ Qs\ [] = [[]]$   
 $\mid collapse\text{-state-list } Qn\ Qs\ (q \# qs) = (\text{let } rec = \text{collapse-state-list } Qn\ Qs\ qs \text{ in}$   
 $\text{if } q \mid\in\mid Qn \wedge q \mid\in\mid Qs \text{ then } map\ (Cons\ None)\ rec\ @\ map\ (Cons\ (Some\ q))\ rec$   
 $\text{else if } q \mid\in\mid Qn \text{ then } map\ (Cons\ None)\ rec$   
 $\text{else if } q \mid\in\mid Qs \text{ then } map\ (Cons\ (Some\ q))\ rec$   
 $\text{else } [[]])$

**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**  $\text{length } (qss ! i) = \text{length } qs \langle \text{proof} \rangle$

**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**  $qs' \in \text{fset-of-list } (\text{collapse-state-list } Qn \ Qs \ qs) \langle \text{proof} \rangle$

**lemma** *collapse-fset-inv-constr2*:

**assumes**  $\forall i < \text{length } qs. qs ! i \in Qn \vee qs ! i \in Qs$   
**and**  $qs' \in \text{fset-of-list } (\text{collapse-state-list } Qn \ Qs \ qs)$  **and**  $i < \text{length } qs'$   
**shows**  $qs ! i \in Qn \wedge qs' ! i = \text{None} \vee qs ! i \in Qs \wedge qs' ! i = \text{Some } (qs ! i)$   
 $\langle \text{proof} \rangle$

**definition** *collapse-rule where*

$\text{collapse-rule } A \ Qn \ Qs =$   
 $|\cup| ((\lambda r. \text{fset-of-list } (\text{map } (\lambda qs. \text{TA-rule } (r\text{-root } r) \ qs \ (\text{Some } (r\text{-rhs } r))))$   
 $(\text{collapse-state-list } Qn \ Qs \ (r\text{-lhs-states } r)))) \ |^{\dagger}$   
 $\text{ffilter } (\lambda r. (\forall i < \text{length } (r\text{-lhs-states } r). r\text{-lhs-states } r ! i \in Qn \vee r\text{-lhs-states}$   
 $r ! i \in Qs))$   
 $(\text{filter } (\lambda r. r\text{-root } r \neq \text{None}) (\text{rules } A))$

**definition** *collapse-rule-fset where*

$\text{collapse-rule-fset } A \ Qn \ Qs = (\lambda r. \text{TA-rule } (r\text{-root } r) (\text{map the } (\text{filter } (\lambda q.$   
 $\neg \text{Option.is-none } q) (r\text{-lhs-states } r))) (\text{the } (r\text{-rhs } r))) \ |^{\dagger}$   
 $\text{collapse-rule } A \ Qn \ Qs$

**lemma** *collapse-rule-set-conv*:

$\text{fset } (\text{collapse-rule-fset } A \ Qn \ Qs) = \{\text{TA-rule } f (\text{map the } (\text{filter } (\lambda q. \neg \text{Option.is-none } q)$   
 $qs')) \ q \mid f \ qs \ qs' \ q.$   
 $\text{TA-rule } (\text{Some } f) \ qs \ q \in \text{rules } A \wedge \text{length } qs = \text{length } qs' \wedge$   
 $(\forall i < \text{length } qs. qs ! i \in Qn \wedge qs' ! i = \text{None} \vee qs ! i \in Qs \wedge (qs' ! i) =$   
 $\text{Some } (qs ! i))\}$  **(is ?Ls = ?Rs)**  
 $\langle \text{proof} \rangle$

**lemma** *collapse-rule-fmember [simp]*:

$\text{TA-rule } f \ qs \ q \in (\text{collapse-rule-fset } A \ Qn \ Qs) \iff (\exists qs' ps.$   
 $qs = \text{map the } (\text{filter } (\lambda q. \neg \text{Option.is-none } q) \ qs') \wedge \text{TA-rule } (\text{Some } f) \ ps \ q \in$   
 $\text{rules } A \wedge \text{length } ps = \text{length } qs' \wedge$   
 $(\forall i < \text{length } ps. ps ! i \in Qn \wedge qs' ! i = \text{None} \vee ps ! i \in Qs \wedge (qs' ! i) = \text{Some}$   
 $(ps ! i)))$   
 $\langle \text{proof} \rangle$

**definition**  $Qn \ A \equiv (\text{let } S = (r\text{-rhs } |^{\dagger} \text{ffilter } (\lambda r. r\text{-root } r = \text{None}) (\text{rules } A)) \text{ in}$

$(\text{eps } A)^+ | \cdot^q S | \cup S$

**definition**  $Qs\ A \equiv (\text{let } S = (\text{r-rhs } | \cdot^q \text{ffilter } (\lambda r. \text{r-root } r \neq \text{None}) (\text{rules } A)) \text{ in } (\text{eps } A)^+ | \cdot^q S | \cup S)$

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

$q \in | Qn\ A \longleftrightarrow (\exists ps\ p. \text{TA-rule None } ps\ p \in | \text{rules } A \wedge (p = q \vee (p, q) \in | (\text{eps } A)^+ |))$  (is ?Ls  $\longleftrightarrow$  ?Rs)  
<proof>

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

$q \in | Qs\ A \longleftrightarrow (\exists f\ ps\ p. \text{TA-rule (Some } f) ps\ p \in | \text{rules } A \wedge (p = q \vee (p, q) \in | (\text{eps } A)^+ |))$  (is ?Ls  $\longleftrightarrow$  ?Rs)  
<proof>

**lemma**  $\text{collapse-}Qn\text{-}Qs\text{-set-conv}$ :

$\text{fset } (Qn\ A) = \{q' \mid qs\ q\ q'. \text{TA-rule None } qs\ q \in | \text{rules } A \wedge (q = q' \vee (q, q') \in | (\text{eps } A)^+ |)\}$  (is ?Ls1 = ?Rs1)  
 $\text{fset } (Qs\ A) = \{q' \mid f\ qs\ q\ q'. \text{TA-rule (Some } f) qs\ q \in | \text{rules } A \wedge (q = q' \vee (q, q') \in | (\text{eps } A)^+ |)\}$  (is ?Ls2 = ?Rs2)  
<proof>

**definition**  $\text{collapse-automaton} :: ('q, 'f \text{ option}) \text{ ta} \Rightarrow ('q, 'f) \text{ ta}$  **where**

$\text{collapse-automaton } A = \text{TA } (\text{collapse-rule-fset } A (Qn\ A) (Qs\ A)) (\text{eps } A)$

**definition**  $\text{collapse-automaton-reg}$  **where**

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

**lemma**  $\text{ta-states-collapse-automaton}$ :

$Q (\text{collapse-automaton } A) \subseteq | Q\ A$   
<proof>

**lemma**  $\text{last-nthI}$ :

**assumes**  $i < \text{length } ts \neg i < \text{length } ts - \text{Suc } 0$   
**shows**  $ts ! i = \text{last } ts$  <proof>

**lemma**  $\text{collapse-automaton}'$ :

**assumes**  $Q\ A \subseteq | \text{ta-reachable } A$   
**shows**  $\text{gta-lang } Q (\text{collapse-automaton } A) = \text{the } \cdot (\text{gcollapse } \cdot \text{gta-lang } Q\ A - \{None\})$   
<proof>

**lemma**  $\mathcal{L}\text{-collapse-automaton}'$ :

**assumes**  $Q_r\ A \subseteq | \text{ta-reachable } (\text{ta } A)$   
**shows**  $\mathcal{L} (\text{collapse-automaton-reg } A) = \text{the } \cdot (\text{gcollapse } \cdot \mathcal{L}\ A - \{None\})$   
<proof>

**lemma**  $\text{collapse-automaton}$ :

**assumes**  $Q_r\ A \subseteq | \text{ta-reachable } (\text{ta } A)$   $RR1\text{-spec } A\ T$

**shows**  $RR1\text{-spec}$  ( $\text{collapse-automaton-reg } A$ ) (the ‘ ( $\text{gcollapse ' } \mathcal{L} A - \{None\}$ ))  
 ⟨proof⟩

## 5.9 Cylindrification

**definition**  $\text{pad-with-Nones}$  **where**

$\text{pad-with-Nones } n m = (\lambda(f, g). \text{case-option } (\text{replicate } n \text{ None}) \text{id } f @ \text{case-option } (\text{replicate } m \text{ None}) \text{id } g)$

**lemma**  $\text{gencode-append}$ :

$\text{gencode } (ss @ ts) = \text{map-gterm } (\text{pad-with-Nones } (\text{length } ss) (\text{length } ts)) (\text{gpair } (\text{gencode } ss) (\text{gencode } ts))$   
 ⟨proof⟩

**lemma**  $\text{append-automaton}$ :

**assumes**  $RRn\text{-spec } n A T RRn\text{-spec } m B U$

**shows**  $RRn\text{-spec } (n + m) (\text{fmap-funs-reg } (\text{pad-with-Nones } n m) (\text{pair-automaton-reg } A B)) \{ts @ us \mid ts \text{ us. } ts \in T \wedge us \in U\}$   
 ⟨proof⟩

**lemma**  $\text{cons-automaton}$ :

**assumes**  $RR1\text{-spec } A T RRn\text{-spec } m B U$

**shows**  $RRn\text{-spec } (\text{Suc } m) (\text{fmap-funs-reg } (\lambda(f, g). \text{pad-with-Nones } 1 m (\text{map-option } (\lambda f. [\text{Some } f]) f, g)) (\text{pair-automaton-reg } A B)) \{t \# us \mid t \text{ us. } t \in T \wedge us \in U\}$   
 ⟨proof⟩

## 5.10 Projection

**abbreviation**  $\text{drop-none-rule } m fs \equiv \text{if list-all } (\text{Option.is-none}) (\text{drop } m fs) \text{ then None else Some } (\text{drop } m fs)$

**lemma**  $\text{drop-automaton-reg}$ :

**assumes**  $\mathcal{Q}_r A \mid \subseteq \mid \text{ta-reachable } (ta A) m < n RRn\text{-spec } n A T$

**defines**  $f \equiv \lambda fs. \text{drop-none-rule } m fs$

**shows**  $RRn\text{-spec } (n - m) (\text{collapse-automaton-reg } (\text{fmap-funs-reg } f A)) (\text{drop } m \text{ ' } T)$   
 ⟨proof⟩

**lemma**  $\text{gfst-collapse-simp}$ :

the  $(\text{gcollapse } (\text{map-gterm } \text{fst } t)) = \text{gfst } t$

⟨proof⟩

**lemma**  $\text{gsnd-collapse-simp}$ :

the  $(\text{gcollapse } (\text{map-gterm } \text{snd } t)) = \text{gsnd } t$

⟨proof⟩

**definition**  $\text{proj-1-reg}$  **where**

$\text{proj-1-reg } A = \text{collapse-automaton-reg } (\text{fmap-funs-reg } \text{fst } (\text{trim-reg } A))$

**definition**  $\text{proj-2-reg}$  **where**

$proj\text{-}2\text{-}reg\ A = collapse\text{-}automaton\text{-}reg\ (fmap\text{-}funs\text{-}reg\ snd\ (trim\text{-}reg\ A))$

**lemmas**  $proj\text{-}1\text{-}reg\text{-}simp = proj\text{-}1\text{-}reg\text{-}def\ collapse\text{-}automaton\text{-}reg\text{-}def\ fmap\text{-}funs\text{-}reg\text{-}def\ trim\text{-}reg\text{-}def$

**lemmas**  $proj\text{-}2\text{-}reg\text{-}simp = proj\text{-}2\text{-}reg\text{-}def\ collapse\text{-}automaton\text{-}reg\text{-}def\ fmap\text{-}funs\text{-}reg\text{-}def\ trim\text{-}reg\text{-}def$

**lemma**  $\mathcal{L}\text{-}proj\text{-}1\text{-}reg\text{-}collapse$ :

$\mathcal{L}\ (proj\text{-}1\text{-}reg\ \mathcal{A}) = the\ ' (gcollapse\ ' map\text{-}gterm\ fst\ ' (\mathcal{L}\ \mathcal{A}) - \{None\})$   
 $\langle proof \rangle$

**lemma**  $\mathcal{L}\text{-}proj\text{-}2\text{-}reg\text{-}collapse$ :

$\mathcal{L}\ (proj\text{-}2\text{-}reg\ \mathcal{A}) = the\ ' (gcollapse\ ' map\text{-}gterm\ snd\ ' (\mathcal{L}\ \mathcal{A}) - \{None\})$   
 $\langle proof \rangle$

**lemma**  $proj\text{-}1$ :

**assumes**  $RR2\text{-}spec\ A\ R$

**shows**  $RR1\text{-}spec\ (proj\text{-}1\text{-}reg\ A)\ (fst\ ' R)$

$\langle proof \rangle$

**lemma**  $proj\text{-}2$ :

**assumes**  $RR2\text{-}spec\ A\ R$

**shows**  $RR1\text{-}spec\ (proj\text{-}2\text{-}reg\ A)\ (snd\ ' R)$

$\langle proof \rangle$

**lemma**  $\mathcal{L}\text{-}proj$ :

**assumes**  $RR2\text{-}spec\ A\ R$

**shows**  $\mathcal{L}\ (proj\text{-}1\text{-}reg\ A) = gfst\ ' \mathcal{L}\ A\ \mathcal{L}\ (proj\text{-}2\text{-}reg\ A) = gsnd\ ' \mathcal{L}\ A$

$\langle proof \rangle$

**lemmas**  $proj\text{-}automaton\text{-}gta\text{-}lang = proj\text{-}1\ proj\text{-}2$

## 5.11 Permutation

**lemma**  $gencode\text{-}permute$ :

**assumes**  $set\ ps = \{0..<length\ ts\}$

**shows**  $gencode\ (map\ (!)\ ts)\ ps = map\text{-}gterm\ (\lambda xs.\ map\ (!)\ xs)\ ps\ (gencode\ ts)$

$\langle proof \rangle$

**lemma**  $permute\text{-}automaton$ :

**assumes**  $RRn\text{-}spec\ n\ A\ T\ set\ ps = \{0..<n\}$

**shows**  $RRn\text{-}spec\ (length\ ps)\ (fmap\text{-}funs\text{-}reg\ (\lambda xs.\ map\ (!)\ xs)\ ps)\ A\ ((\lambda xs.\ map\ (!)\ xs)\ ps)\ ' T$

$\langle proof \rangle$

## 5.12 Intersection

**lemma**  $intersect\text{-}automaton$ :

**assumes**  $RRn\text{-}spec\ n\ A\ T\ RRn\text{-}spec\ n\ B\ U$

**shows**  $RRn\text{-spec } n \text{ (reg-intersect } A \ B) \ (T \cap U) \langle \text{proof} \rangle$

**lemma** *union-automaton*:

**assumes**  $RRn\text{-spec } n \ A \ T \ RRn\text{-spec } n \ B \ U$   
**shows**  $RRn\text{-spec } n \text{ (reg-union } A \ B) \ (T \cup U)$   
 $\langle \text{proof} \rangle$

### 5.13 Difference

**lemma** *RR1-difference*:

**assumes**  $RR1\text{-spec } A \ T \ RR1\text{-spec } B \ U$   
**shows**  $RR1\text{-spec } \text{(difference-reg } A \ B) \ (T - U)$   
 $\langle \text{proof} \rangle$

**lemma** *RR2-difference*:

**assumes**  $RR2\text{-spec } A \ T \ RR2\text{-spec } B \ U$   
**shows**  $RR2\text{-spec } \text{(difference-reg } A \ B) \ (T - U)$   
 $\langle \text{proof} \rangle$

**lemma** *RRn-difference*:

**assumes**  $RRn\text{-spec } n \ A \ T \ RRn\text{-spec } n \ B \ U$   
**shows**  $RRn\text{-spec } n \ \text{(difference-reg } A \ B) \ (T - U)$   
 $\langle \text{proof} \rangle$

### 5.14 All terms over a signature

**definition** *term-automaton* ::  $(f \times \text{nat}) \text{ fset} \Rightarrow (\text{unit}, f) \text{ ta}$  **where**

$\text{term-automaton } \mathcal{F} = \text{TA } ((\lambda (f, n). \text{TA-rule } f \text{ (replicate } n \ ()) \ ()) \mid \mathcal{F}) \{\|\}$

**definition** *term-reg* **where**

$\text{term-reg } \mathcal{F} = \text{Reg } \{\{()\}\} \text{ (term-automaton } \mathcal{F})$

**lemma** *term-automaton*:

$RR1\text{-spec } \text{(term-reg } \mathcal{F}) \ (\mathcal{T}_G \text{ (fset } \mathcal{F}))$   
 $\langle \text{proof} \rangle$

**fun** *true-RRn* ::  $(f \times \text{nat}) \text{ fset} \Rightarrow \text{nat} \Rightarrow (\text{nat}, f \text{ option list}) \text{ reg}$  **where**

$\text{true-RRn } \mathcal{F} \ 0 = \text{Reg } \{\{0\}\} \text{ (TA } \{\{\text{TA-rule } [] \ [] \ 0\}\} \{\|\})$   
 $\mid \text{true-RRn } \mathcal{F} \ (\text{Suc } 0) = \text{relabel-reg } \text{(fmap-funs-reg } (\lambda f. [\text{Some } f]) \text{ (term-reg } \mathcal{F}))$   
 $\mid \text{true-RRn } \mathcal{F} \ (\text{Suc } n) = \text{relabel-reg}$   
 $\text{(trim-reg (fmap-funs-reg (pad-with-Nones } 1 \ n) \text{ (pair-automaton-reg (true-RRn } \mathcal{F} \ 1) \text{ (true-RRn } \mathcal{F} \ n))))$

**lemma** *true-RRn-spec*:

$RRn\text{-spec } n \ \text{(true-RRn } \mathcal{F} \ n) \ \{\text{ts. length } \text{ts} = n \wedge \text{set } \text{ts} \subseteq \mathcal{T}_G \text{ (fset } \mathcal{F})\}$   
 $\langle \text{proof} \rangle$

## 5.15 RR2 composition

**abbreviation** *RR2-to-RRn*  $A \equiv \text{fmap-funs-reg } (\lambda(f, g). [f, g]) A$

**abbreviation** *RRn-to-RR2*  $A \equiv \text{fmap-funs-reg } (\lambda f. (f ! 0, f ! 1)) A$

**definition** *rr2-compositon* **where**

*rr2-compositon*  $\mathcal{F} A B =$   
 (let  $A' = \text{RR2-to-RRn } A$  in  
 let  $B' = \text{RR2-to-RRn } B$  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  
 $\text{RRn-to-RR2 } (\text{collapse-automaton-reg } (\text{fmap-funs-reg } (\text{drop-none-rule } 1) PI))$   
 )

**lemma** *list-length1E*:

**assumes**  $\text{length } xs = \text{Suc } 0$  **obtains**  $x$  **where**  $xs = [x]$   $\langle \text{proof} \rangle$

**lemma** *rr2-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-spec*  $A \mathcal{R}$  **and** *RR2-spec*  $B \mathcal{L}$

**shows** *RR2-spec*  $(\text{rr2-compositon } \mathcal{F} A B) (\mathcal{R} \circ \mathcal{L})$

$\langle \text{proof} \rangle$

**end**

**theory** *RR2-Infinite*

**imports** *RRn-Automata Tree-Automata-Pumping*

**begin**

**lemma** *map-ta-rule-id* [*simp*]:  $\text{map-ta-rule } f \text{ id } r = (\text{r-root } r) (\text{map } f (\text{r-lhs-states } r)) \rightarrow (f (\text{r-rhs } r))$  **for**  $f r$

$\langle \text{proof} \rangle$

**lemma** *no-upper-bound-infinite*:

**assumes**  $\forall (n::\text{nat}). \exists t \in S. n < f t$

**shows** *infinite*  $S$

$\langle \text{proof} \rangle$

**lemma** *set-constr-finite*:

**assumes** *finite*  $F$

**shows** *finite*  $\{h x \mid x. x \in F \wedge P x\}$   $\langle \text{proof} \rangle$

**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*  $S$   $\langle \text{proof} \rangle$

**lemma** *infinite-imageD*:  
*infinite*  $(f \text{ ' } S) \implies \text{inj-on } f \ S \implies \text{infinite } S$   
 $\langle \text{proof} \rangle$

**lemma** *infinite-imageD2*:  
*infinite*  $(f \text{ ' } S) \implies \text{inj } f \implies \text{infinite } S$   
 $\langle \text{proof} \rangle$

**lemma** *infinite-inj-image-infinite*:  
**assumes** *infinite*  $S$  **and** *inj-on*  $f \ S$   
**shows** *infinite*  $(f \text{ ' } S)$   
 $\langle \text{proof} \rangle$

**lemma** *infinte-no-depth-limit*:  
**assumes** *infinite*  $S$  **and** *finite*  $\mathcal{F}$   
**and**  $\forall t \in S. \text{funas-term } t \subseteq \mathcal{F}$  **and**  $\forall t \in S. \text{ground } t$   
**shows**  $\forall (n::\text{nat}). \exists t \in S. n < (\text{depth } t)$   
 $\langle \text{proof} \rangle$

**lemma** *depth-gterm-conv*:  
 $\text{depth } (\text{term-of-gterm } t) = \text{depth } (\text{term-of-gterm } t)$   
 $\langle \text{proof} \rangle$

**lemma** *funas-term-ctxt [simp]*:  
 $\text{funas-term } C \langle s \rangle = \text{funas-ctxt } C \cup \text{funas-term } s$   
 $\langle \text{proof} \rangle$

**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*  $\{t . q \in | \ \text{gta-der } \mathcal{A} \ t \wedge P \ (\text{funas-gterm } t)\}$   
 $\langle \text{proof} \rangle$

**lemma** *gterm-to-None-Some-funas [simp]*:  
 $\text{funas-gterm } (\text{gterm-to-None-Some } t) \subseteq (\lambda (f, n). ((\text{None}, \text{Some } f), n)) \text{ ' } \mathcal{F} \longleftrightarrow$   
 $\text{funas-gterm } t \subseteq \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *funas-gterm-bot-some-decomp*:  
**assumes**  $\text{funas-gterm } s \subseteq (\lambda (f, n). ((\text{None}, \text{Some } f), n)) \text{ ' } \mathcal{F}$   
**shows**  $\exists t. \text{gterm-to-None-Some } t = s \wedge \text{funas-gterm } t \subseteq \mathcal{F}$   $\langle \text{proof} \rangle$

**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*  $\mathcal{A} \mathcal{F} = \{|q | q. \text{infinite } \{t | t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q | \in | \text{ta-der } \mathcal{A} (\text{term-of-gterm } (\text{gterm-to-None-Some } t))\}| \}$

**lemma** *Q-infty-fmember*:

$q | \in | \text{Q-infty } \mathcal{A} \mathcal{F} \longleftrightarrow \text{infinite } \{t | t. \text{funas-gterm } t \subseteq \text{fset } \mathcal{F} \wedge q | \in | \text{ta-der } \mathcal{A} (\text{term-of-gterm } (\text{gterm-to-None-Some } t))\}$   
*<proof>*

**abbreviation** *q-inf-dash-intro-rules* **where**

*q-inf-dash-intro-rules*  $Q r \equiv \text{if } (r\text{-rhs } r) | \in | Q \wedge \text{fst } (r\text{-root } r) = \text{None} \text{ then } \{(r\text{-root } r) (\text{map } \text{CInl } (r\text{-lhs-states } r)) \rightarrow \text{CInr } (r\text{-rhs } r)\} \text{ else } \{\}\}$

**abbreviation** *args*  $:: 'a \text{ list} \Rightarrow \text{nat} \Rightarrow ('a + 'a) \text{ list}$  **where**

*args*  $\equiv \lambda \text{qs } i. \text{map } \text{CInl } (\text{take } i \text{ qs}) @ \text{CInr } (\text{qs } ! i) \# \text{map } \text{CInl } (\text{drop } (\text{Suc } i) \text{ qs})$

**abbreviation** *q-inf-dash-closure-rules*  $:: ('q, 'f) \text{ ta-rule} \Rightarrow ('q + 'q, 'f) \text{ ta-rule list}$  **where**

*q-inf-dash-closure-rules*  $r \equiv (\text{let } (f, \text{qs}, q) = (r\text{-root } r, r\text{-lhs-states } r, r\text{-rhs } r) \text{ in } (\text{map } (\lambda i. f (\text{args } \text{qs } i)) \rightarrow \text{CInr } q) [0 .. < \text{length } \text{qs}])$

**definition** *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**

*Inf-automata*  $\mathcal{A} Q = \text{TA}$   
 $((| \cup | (\text{q-inf-dash-intro-rules } Q | \uparrow \text{rules } \mathcal{A})) | \cup | (| \cup | ((\text{fset-of-list} \circ \text{q-inf-dash-closure-rules}) | \uparrow \text{rules } \mathcal{A})) | \cup | \text{map-ta-rule } \text{CInl } \text{id } | \uparrow \text{rules } \mathcal{A}) (\text{map-both } \text{Inl } | \uparrow \text{eps } \mathcal{A} | \cup | \text{map-both } \text{CInr } | \uparrow \text{eps } \mathcal{A}))$

**definition** *Inf-reg* **where**

*Inf-reg*  $\mathcal{A} Q = \text{Reg } (\text{CInr } | \uparrow \text{fin } \mathcal{A}) (\text{Inf-automata } (\text{ta } \mathcal{A}) Q)$

**lemma** *Inr-Inl-rel-comp*:

*map-both CInr | \uparrow S | O | map-both CInl | \uparrow S = \{\}\}* *<proof>*

**lemmas** *eps-split* = *ftrancl-Un2-separatorE[OF Inr-Inl-rel-comp]*

**lemma** *Inf-automata-eps-simp* [*simp*]:

**shows**  $(\text{map-both } \text{Inl } | \uparrow \text{eps } \mathcal{A} | \cup | \text{map-both } \text{CInr } | \uparrow \text{eps } \mathcal{A})^+ = (\text{map-both } \text{CInl } | \uparrow \text{eps } \mathcal{A})^+ | \cup | (\text{map-both } \text{CInr } | \uparrow \text{eps } \mathcal{A})^+$   
*<proof>*

**lemma** *map-both-CInl-ftrancl-conv*:

$(\text{map-both } \text{CInl } | \uparrow \text{eps } \mathcal{A})^+ = \text{map-both } \text{CInl } | \uparrow (\text{eps } \mathcal{A})^+$   
*<proof>*

**lemma** *map-both-CInr-ftrancl-conv*:

$(\text{map-both } CInr \mid^{\dagger} \text{ eps } \mathcal{A})|^{+} = \text{map-both } CInr \mid^{\dagger} (\text{eps } \mathcal{A})|^{+}$   
 $\langle \text{proof} \rangle$

**lemmas** *map-both-ftrancl-conv = map-both-CInl-ftrancl-conv map-both-CInr-ftrancl-conv*

**lemma** *Inf-automata-Inl-to-eps* [simp]:

$(CInl \ p, CInl \ q) \mid \in \mid (\text{map-both } CInl \mid^{\dagger} \text{ eps } \mathcal{A})|^{+} \longleftrightarrow (p, q) \mid \in \mid (\text{eps } \mathcal{A})|^{+}$   
 $(CInr \ p, CInr \ q) \mid \in \mid (\text{map-both } CInr \mid^{\dagger} \text{ eps } \mathcal{A})|^{+} \longleftrightarrow (p, q) \mid \in \mid (\text{eps } \mathcal{A})|^{+}$   
 $(CInl \ q, CInl \ p) \mid \in \mid (\text{map-both } CInr \mid^{\dagger} \text{ eps } \mathcal{A})|^{+} \longleftrightarrow \text{False}$   
 $(CInr \ q, CInr \ p) \mid \in \mid (\text{map-both } CInl \mid^{\dagger} \text{ eps } \mathcal{A})|^{+} \longleftrightarrow \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *Inl-eps-Inr*:

$(CInl \ q, CInl \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+} \longleftrightarrow (CInr \ q, CInr \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+}$   
 $\langle \text{proof} \rangle$

**lemma** *Inr-rhs-eps-Inr-lhs*:

**assumes**  $(q, CInr \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+}$   
**obtains**  $q'$  **where**  $q = CInr \ q'$   $\langle \text{proof} \rangle$

**lemma** *Inl-rhs-eps-Inl-lhs*:

**assumes**  $(q, CInl \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+}$   
**obtains**  $q'$  **where**  $q = CInl \ q'$   $\langle \text{proof} \rangle$

**lemma** *Inf-automata-eps* [simp]:

$(CInl \ q, CInr \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+} \longleftrightarrow \text{False}$   
 $(CInr \ q, CInl \ p) \mid \in \mid (\text{eps } (\text{Inf-automata } \mathcal{A} \ Q))|^{+} \longleftrightarrow \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *Inl-A-res-Inf-automata*:

$\text{ta-der } (\text{fmap-states-ta } CInl \ \mathcal{A}) \ t \mid \subseteq \mid \text{ta-der } (\text{Inf-automata } \mathcal{A} \ Q) \ t$   
 $\langle \text{proof} \rangle$

**lemma** *Inl-res-A-res-Inf-automata*:

$CInl \ \mid^{\dagger} \text{ta-der } \mathcal{A} \ (\text{term-of-gterm } t) \mid \subseteq \mid \text{ta-der } (\text{Inf-automata } \mathcal{A} \ Q) \ (\text{term-of-gterm } t)$   
 $\langle \text{proof} \rangle$

**lemma** *r-rhs-CInl-args-A-rule*:

**assumes**  $f \ q_s \rightarrow CInl \ q \mid \in \mid \text{rules } (\text{Inf-automata } \mathcal{A} \ Q)$   
**obtains**  $q_s'$  **where**  $q_s = \text{map } CInl \ q_s' \ f \ q_s' \rightarrow q \mid \in \mid \text{rules } \mathcal{A}$   $\langle \text{proof} \rangle$

**lemma** *A-rule-to-dash-closure*:

**assumes**  $f \ q_s \rightarrow q \mid \in \mid \text{rules } \mathcal{A}$  **and**  $i < \text{length } q_s$   
**shows**  $f \ (\text{args } q_s \ i) \rightarrow CInr \ q \mid \in \mid \text{rules } (\text{Inf-automata } \mathcal{A} \ Q)$   
 $\langle \text{proof} \rangle$

**lemma** *Inf-automata-reach-to-dash-reach:*

**assumes**  $CInl\ p \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ C\langle Var\ (CInl\ q)\rangle$   
**shows**  $CInr\ p \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ C\langle Var\ (CInr\ q)\rangle$  (**is** -  $\mid \in \mid ta\text{-der}\ ?A\ -$ )  
 $\langle proof \rangle$

**lemma** *Inf-automata-dashI:*

**assumes**  $run\ \mathcal{A}\ r\ (gterm\text{-to}\ None\ Some\ t)$  **and**  $ex\text{-rule}\text{-state}\ r \mid \in \mid Q$   
**shows**  $CInr\ (ex\text{-rule}\text{-state}\ r) \mid \in \mid gta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ (gterm\text{-to}\ None\ Some\ t)$   
 $\langle proof \rangle$

**lemma** *Inf-automata-dash-reach-to-reach:*

**assumes**  $p \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ t$  (**is** -  $\mid \in \mid ta\text{-der}\ ?A\ -$ )  
**shows**  $remove\text{-sum}\ p \mid \in \mid ta\text{-der}\ \mathcal{A}\ (map\text{-vars}\text{-term}\ remove\text{-sum}\ t)$   $\langle proof \rangle$

**lemma** *depth-poss-split:*

**assumes**  $Suc\ (depth\ (term\text{-of}\ gterm\ t) + n) < depth\ (term\text{-of}\ gterm\ u)$   
**shows**  $\exists\ p\ q.\ p\ @\ q \in\ gposs\ u \wedge n < length\ q \wedge p \notin gposs\ t$   
 $\langle proof \rangle$

**lemma** *Inf-to-automata:*

**assumes**  $RR2\text{-spec}\ \mathcal{A}\ \mathcal{R}$  **and**  $t \in Inf\text{-branching}\text{-terms}\ \mathcal{R}\ \mathcal{F}$   
**shows**  $\exists\ u.\ gpair\ t\ u \in \mathcal{L}\ (Inf\text{-reg}\ \mathcal{A}\ (Q\text{-infty}\ (ta\ \mathcal{A})\ \mathcal{F}))$  (**is**  $\exists\ u.\ gpair\ t\ u \in \mathcal{L}\ ?B$ )  
 $\langle proof \rangle$

**lemma** *CInr-Inf-automata-to-q-state:*

**assumes**  $CInr\ p \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ t$  **and**  $ground\ t$   
**shows**  $\exists\ C\ s\ q.\ C\langle s \rangle = t \wedge CInr\ q \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ s \wedge q \mid \in \mid Q \wedge$   
 $CInr\ p \mid \in \mid ta\text{-der}\ (Inf\text{-automata}\ \mathcal{A}\ Q)\ C\langle Var\ (CInr\ q)\rangle \wedge$   
 $(fst \circ fst \circ the \circ root)\ s = None$   $\langle proof \rangle$

**lemma** *aux-lemma:*

**assumes**  $RR2\text{-spec}\ \mathcal{A}\ \mathcal{R}$  **and**  $\mathcal{R} \subseteq \mathcal{T}_G\ (fset\ \mathcal{F}) \times \mathcal{T}_G\ (fset\ \mathcal{F})$   
**and**  $infinite\ \{u \mid u.\ gpair\ t\ u \in \mathcal{L}\ \mathcal{A}\}$   
**shows**  $t \in Inf\text{-branching}\text{-terms}\ \mathcal{R}\ \mathcal{F}$   
 $\langle proof \rangle$

**lemma** *Inf-automata-to-Inf:*

**assumes**  $RR2\text{-spec}\ \mathcal{A}\ \mathcal{R}$  **and**  $\mathcal{R} \subseteq \mathcal{T}_G\ (fset\ \mathcal{F}) \times \mathcal{T}_G\ (fset\ \mathcal{F})$   
**and**  $gpair\ t\ u \in \mathcal{L}\ (Inf\text{-reg}\ \mathcal{A}\ (Q\text{-infty}\ (ta\ \mathcal{A})\ \mathcal{F}))$   
**shows**  $t \in Inf\text{-branching}\text{-terms}\ \mathcal{R}\ \mathcal{F}$   
 $\langle proof \rangle$

**lemma** *Inf-automata-subseteq:*

$\mathcal{L}\ (Inf\text{-reg}\ \mathcal{A}\ (Q\text{-infty}\ (ta\ \mathcal{A})\ \mathcal{F})) \subseteq \mathcal{L}\ \mathcal{A}$  (**is**  $\mathcal{L}\ ?IA \subseteq -$ )  
 $\langle proof \rangle$

**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**  $\text{gfst } \mathcal{L} (\text{Inf-reg } \mathcal{A} (Q\text{-infty } (ta \ \mathcal{A}) \ \mathcal{F})) = \text{Inf-branching-terms } \mathcal{R} \ \mathcal{F}$   
 $\langle \text{proof} \rangle$   
**end**  
**theory** *Tree-Automata-Abstract-Impl*  
**imports** *Tree-Automata-Det Horn-Fset*  
**begin**

## 6 Computing state derivation

**lemma** *ta-der-Var-code* [code]:  
 $ta\text{-der } \mathcal{A} (\text{Var } q) = \text{finsert } q ((\text{eps } \mathcal{A})^+ | \{q\})$   
 $\langle \text{proof} \rangle$

**lemma** *ta-der-Fun-code* [code]:  
 $ta\text{-der } \mathcal{A} (\text{Fun } f \ ts) =$   
 $(\text{let } args = \text{map } (ta\text{-der } \mathcal{A}) \ ts \ \text{in}$   
 $\text{let } P = (\lambda r. \text{case } r \ \text{of } TA\text{-rule } g \ ps \ p \Rightarrow f = g \wedge \text{list-all2 } f\text{member } ps \ args) \ \text{in}$   
 $\text{let } S = r\text{-rhs } | \text{ffilter } P \ (\text{rules } \mathcal{A}) \ \text{in}$   
 $S \ | \cup \ | (\text{eps } \mathcal{A})^+ | \{S\} \ (\text{is } ?Ls = ?Rs)$   
 $\langle \text{proof} \rangle$

**definition** *eps-free-automata where*  
 $\text{eps-free-automata } \text{epscl } \mathcal{A} =$   
 $(\text{let } ruleps = (\lambda r. \text{finsert } (r\text{-rhs } r) (\text{epscl } | \{r\text{-rhs } r\})) \ \text{in}$   
 $\text{let } rules = (\lambda r. (\lambda q. TA\text{-rule } (r\text{-root } r) (r\text{-lhs-states } r) q) | \text{ruleps } r) | \text{rules } \mathcal{A}) \ \text{in}$   
 $TA \ (| \cup \ | rules) \ \{\}\}$

**lemma** *eps-free* [code]:  
 $\text{eps-free } \mathcal{A} = \text{eps-free-automata } ((\text{eps } \mathcal{A})^+ |) \ \mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma** *eps-of-eps-free-automata* [simp]:  
 $\text{eps } (\text{eps-free-automata } S \ \mathcal{A}) = \{\}\}$   
 $\langle \text{proof} \rangle$

**lemma** *eps-free-automata-empty* [simp]:  
 $\text{eps } \mathcal{A} = \{\}\} \Longrightarrow \text{eps-free-automata } \{\}\} \ \mathcal{A} = \mathcal{A}$   
 $\langle \text{proof} \rangle$

## 7 Computing the restriction of tree automata to state set

**lemma** *ta-restrict* [code]:

$ta\text{-restrict } \mathcal{A} \ Q =$   
 $(let\ rules = ffilter\ (\lambda\ r.\ case\ r\ of\ TA\text{-rule}\ f\ ps\ p \Rightarrow fset\text{-of-list}\ ps \subseteq | Q \wedge p$   
 $|\in| Q) (rules\ \mathcal{A})\ in$   
 $let\ eps = ffilter\ (\lambda\ r.\ case\ r\ of\ (p, q) \Rightarrow p |\in| Q \wedge q |\in| Q) (eps\ \mathcal{A})\ in$   
 $TA\ rules\ eps)$   
 $\langle proof \rangle$

## 8 Computing the epsilon transition for the product automaton

**lemma** *prod-eps[code-unfold]*:

$fCollect\ (prod\text{-eps}Lp\ \mathcal{A}\ \mathcal{B}) = (\lambda\ ((p, q), r). ((p, r), (q, r))) |^+ (eps\ \mathcal{A} \times | \mathcal{Q}\ \mathcal{B})$   
 $fCollect\ (prod\text{-eps}Rp\ \mathcal{A}\ \mathcal{B}) = (\lambda\ ((p, q), r). ((r, p), (r, q))) |^+ (eps\ \mathcal{B} \times | \mathcal{Q}\ \mathcal{A})$   
 $\langle proof \rangle$

## 9 Computing reachability

**inductive-set** *ta-reach* for  $\mathcal{A}$  where

$rule\ [intro]: f\ qs \rightarrow q |\in| rules\ \mathcal{A} \Longrightarrow \forall\ i < length\ qs.\ qs\ !\ i \in ta\text{-reach}\ \mathcal{A} \Longrightarrow q$   
 $\in ta\text{-reach}\ \mathcal{A}$   
 $| eps\ [intro]: q \in ta\text{-reach}\ \mathcal{A} \Longrightarrow (q, r) |\in| eps\ \mathcal{A} \Longrightarrow r \in ta\text{-reach}\ \mathcal{A}$

**lemma** *ta-reach-eps-transI*:

**assumes**  $(p, q) |\in| (eps\ \mathcal{A})|^+ | p \in ta\text{-reach}\ \mathcal{A}$   
**shows**  $q \in ta\text{-reach}\ \mathcal{A}$   $\langle proof \rangle$

**lemma** *ta-reach-ground-term-der*:

**assumes**  $q \in ta\text{-reach}\ \mathcal{A}$   
**shows**  $\exists\ t.\ ground\ t \wedge q |\in| ta\text{-der}\ \mathcal{A}\ t$   $\langle proof \rangle$

**lemma** *ground-term-der-ta-reach*:

**assumes**  $ground\ t\ q |\in| ta\text{-der}\ \mathcal{A}\ t$   
**shows**  $q \in ta\text{-reach}\ \mathcal{A}$   $\langle proof \rangle$

**lemma** *ta-reach-reachable*:

$ta\text{-reach}\ \mathcal{A} = fset\ (ta\text{-reachable}\ \mathcal{A})$   
 $\langle proof \rangle$

### 9.1 Horn setup for reachable states

**definition** *reach-rules*  $\mathcal{A} =$

$\{qs \rightarrow_h q \mid f\ qs\ q.\ TA\text{-rule}\ f\ qs\ q \mid \in | rules\ \mathcal{A}\} \cup$   
 $\{[q] \rightarrow_h r \mid q\ r.\ (q, r) |\in| eps\ \mathcal{A}\}$

**locale** *reach-horn* =

**fixes**  $\mathcal{A} :: ('q, 'f)\ ta$   
**begin**

**sublocale** *horn reach-rules*  $\mathcal{A}$   $\langle$ proof $\rangle$

**lemma** *reach-infer0*:  $infer0 = \{q \mid f q. TA\text{-rule } f \ [] \ q \mid \in \mid \text{rules } \mathcal{A}\}$   
 $\langle$ proof $\rangle$

**lemma** *reach-infer1*:

*infer1*  $p \ X = \{r \mid f \ q \ s \ r. TA\text{-rule } f \ q \ s \ r \mid \in \mid \text{rules } \mathcal{A} \wedge p \in \text{set } q \ s \wedge \text{set } q \ s \subseteq \text{insert } p \ X\} \cup$   
 $\{r \mid r. (p, r) \mid \in \mid \text{eps } \mathcal{A}\}$   
 $\langle$ proof $\rangle$

**lemma** *reach-sound*:

*ta-reach*  $\mathcal{A} = \text{saturnate}$   
 $\langle$ proof $\rangle$   
**end**

## 9.2 Computing productivity

First, use an alternative definition of productivity

**inductive-set** *ta-productive-ind* ::  $'q \ \text{fset} \Rightarrow ('q, 'f) \ \text{ta} \Rightarrow 'q \ \text{set for } P \ \text{and } \mathcal{A} ::$   
 $('q, 'f) \ \text{ta}$  **where**

*basic* [*intro*]:  $q \mid \in \mid P \Longrightarrow q \in \text{ta-productive-ind } P \ \mathcal{A}$   
 $\mid \text{eps}$  [*intro*]:  $(p, q) \mid \in \mid (\text{eps } \mathcal{A})^+ \Longrightarrow q \in \text{ta-productive-ind } P \ \mathcal{A} \Longrightarrow p \in \text{ta-productive-ind } P \ \mathcal{A}$   
 $\mid \text{rule}$ :  $TA\text{-rule } f \ q \ s \ q \mid \in \mid \text{rules } \mathcal{A} \Longrightarrow q \in \text{ta-productive-ind } P \ \mathcal{A} \Longrightarrow q' \in \text{set } q \ s$   
 $\Longrightarrow q' \in \text{ta-productive-ind } P \ \mathcal{A}$

**lemma** *ta-productive-ind*:

*ta-productive-ind*  $P \ \mathcal{A} = \text{fset } (\text{ta-productive } P \ \mathcal{A})$  (**is** ?LS = ?RS)  
 $\langle$ proof $\rangle$

### 9.2.1 Horn setup for productive states

**definition** *productive-rules*  $P \ \mathcal{A} = \{\ [] \ \rightarrow_h \ q \mid q. q \mid \in \mid P\} \cup$   
 $\{[r] \ \rightarrow_h \ q \mid q \ r. (q, r) \mid \in \mid \text{eps } \mathcal{A}\} \cup$   
 $\{[q] \ \rightarrow_h \ r \mid f \ q \ s \ q \ r. TA\text{-rule } f \ q \ s \ q \mid \in \mid \text{rules } \mathcal{A} \wedge r \in \text{set } q \ s\}$

**locale** *productive-horn* =

**fixes**  $\mathcal{A} :: ('q, 'f) \ \text{ta}$  **and**  $P :: 'q \ \text{fset}$   
**begin**

**sublocale** *horn productive-rules*  $P \ \mathcal{A}$   $\langle$ proof $\rangle$

**lemma** *productive-infer0*:  $infer0 = \text{fset } P$   
 $\langle$ proof $\rangle$

**lemma** *productive-infer1*:

*infer1*  $p \ X = \{r \mid r. (r, p) \mid \in \mid \text{eps } \mathcal{A}\} \cup$

$\{r \mid f \text{ qs } r. \text{ TA-rule } f \text{ qs } p \mid \in \mid \text{ rules } \mathcal{A} \wedge r \in \text{ set } \text{qs}\}$   
 <proof>

**lemma** *productive-sound*:  
 ta-productive-ind P  $\mathcal{A} = \text{ saturate}$   
 <proof>  
**end**

### 9.3 Horn setup for power set construction states

**lemma** *prod-list-exists*:  
 assumes  $\text{fst } p \in \text{ set } \text{qs}$   $\text{set } \text{qs} \subseteq \text{ insert } (\text{fst } p) (\text{fst } 'X)$   
 obtains  $\text{as}$  **where**  $p \in \text{ set } \text{as}$   $\text{map } \text{fst } \text{as} = \text{qs}$   $\text{set } \text{as} \subseteq \text{ insert } p \text{ X}$   
 <proof>

**definition** *ps-states-rules*  $\mathcal{A} = \{rs \rightarrow_h (\text{Wrapp } q) \mid rs \text{ f } q.$   
 $q = \text{ps-reachable-states } \mathcal{A} \text{ f } (\text{map } \text{ex } rs) \wedge q \neq \{\}\}$

**locale** *ps-states-horn* =  
 fixes  $\mathcal{A} :: ('q, 'f)$  ta  
**begin**

**sublocale** *horn* *ps-states-rules*  $\mathcal{A}$  <proof>

**lemma** *ps-construction-infer0*: *infer0* =  
 $\{\text{Wrapp } q \mid f \text{ q. } q = \text{ps-reachable-states } \mathcal{A} \text{ f } [] \wedge q \neq \{\}\}$   
 <proof>

**lemma** *ps-construction-infer1*:  
 $\text{infer1 } p \text{ X} = \{\text{Wrapp } q \mid f \text{ qs } q. q = \text{ps-reachable-states } \mathcal{A} \text{ f } (\text{map } \text{ex } \text{qs}) \wedge q \neq \{\}\} \wedge$   
 $p \in \text{ set } \text{qs} \wedge \text{ set } \text{qs} \subseteq \text{ insert } p \text{ X}$   
 <proof>

**lemma** *ps-states-sound*:  
 $\text{ps-states-set } \mathcal{A} = \text{ saturate}$   
 <proof>

**end**

**definition** *ps-reachable-states-cont* **where**  
 $\text{ps-reachable-states-cont } \Delta \Delta_\epsilon \text{ f } \text{ps} =$   
 (let  $R = \text{ffilter } (\lambda r. \text{ case } r \text{ of TA-rule } g \text{ qs } q \Rightarrow f = g \wedge \text{list-all2 } (|\in|) \text{ qs } \text{ps}) \Delta$   
 in  
 let  $S = r\text{-rhs } |\cdot| \text{ R}$  in  
 $S \mid \cup \mid \Delta_\epsilon \mid^+ \mid \cdot \mid S)$

**lemma** *ps-reachable-states* [code]:  
 $\text{ps-reachable-states } (\text{TA } \Delta \Delta_\epsilon) \text{ f } \text{ps} = \text{ps-reachable-states-cont } \Delta \Delta_\epsilon \text{ f } \text{ps}$

*<proof>*

**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  $Q$ )))) |q  
sig in  
  let res = ( $\lambda$  ( $f$ ,  $n$ ,  $Qs$ ). ( $\lambda$   $qs$ . TA-rule  $f$   $qs$  (Wrapp (ps-reachable-states  $\mathcal{A}$   $f$  (map  
ex  $qs$ )))) |q  $Qs$ ) |q  $qss$  in  
    ffilter ( $\lambda$   $r$ . ex (r-rhs  $r$ )  $\neq$  {||}) (| $\cup$ | res))
```

**lemma** *ps-rules* [*code*]:

```
ps-rules  $\mathcal{A}$   $Q$  = ps-rules-cont  $\mathcal{A}$   $Q$   
<proof>
```

**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 compare 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  $\Rightarrow$  ' $a$  fset  $\Rightarrow$  order)  
  where compare-fset = ( $\lambda$   $A$   $B$ .  
    (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**

*<proof>*

**end**

**instantiation** *fset* :: (*linorder*) *compare*

**begin**

**definition** *compare-fset* :: ('a *fset* ⇒ 'a *fset* ⇒ *order*) *option*

**where** *compare-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**

⟨*proof*⟩

**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**

⟨*proof*⟩

**end**

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

**begin**

**definition** *compare-FSet-Lex-Wrapper* :: ('a *FSet-Lex-Wrapper* ⇒ 'a *FSet-Lex-Wrapper*  
⇒ *order*) *option*

**where** *compare-FSet-Lex-Wrapper* = *Some* (λ *A B*.

(*let* *A'* = *sorted-list-of-fset* (*ex* *A*) in

*let* *B'* = *sorted-list-of-fset* (*ex* *B*) in

*if* *A'* < *B'* *then* *Lt* *else if* *B'* < *A'* *then* *Gt* *else* *Eq*))

**instance**

⟨*proof*⟩

**end**

**lemma** *infinite-ta-rule-UNIV*[*simp, intro*]: *infinite* (*UNIV* :: ('a,'f) *ta-rule* *set*)  
⟨*proof*⟩

**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**

⟨*proof*⟩

**end**

**instantiation** *ta-rule* :: (*ccompare,ccompare*)*cproper-interval*

**begin**

**definition** *cproper-interval* = ( $\lambda$  ( - :: ('a,'b)*ta-rule option*) - . *False*)

**instance**  $\langle$ *proof* $\rangle$

**end**

**lemma** *finite-finite-Fpow*:

**assumes** *finite A*

**shows** *finite (Fpow A)*  $\langle$ *proof* $\rangle$

**lemma** *infinite-infinite-Fpow*:

**assumes** *infinite A*

**shows** *infinite (Fpow A)*

$\langle$ *proof* $\rangle$

**lemma** *inj-on-Abs-fset*:

$(\bigwedge X. X \in A \implies \text{finite } X) \implies \text{inj-on Abs-fset } A$   $\langle$ *proof* $\rangle$

**lemma** *UNIV-FSet-Lex-Wrapper*:

$(UNIV :: 'a \text{ FSet-Lex-Wrapper set}) = (\text{Wrapp} \circ \text{Abs-fset}) \text{ ` } (\text{Fpow } (UNIV :: 'a \text{ set}))$

$\langle$ *proof* $\rangle$

**lemma** *FSet-Lex-Wrapper-UNIV*:

$(UNIV :: 'a \text{ FSet-Lex-Wrapper set}) = (\text{Wrapp} \circ \text{Abs-fset}) \text{ ` } (\text{Fpow } (UNIV :: 'a \text{ set}))$

$\langle$ *proof* $\rangle$

**lemma** *Wrapp-Abs-fset-inj*:

*inj-on* ( $\text{Wrapp} \circ \text{Abs-fset}$ ) (*Fpow A*)

$\langle$ *proof* $\rangle$

**lemma** *infinite-FSet-Lex-Wrapper-UNIV*:

**assumes** *infinite (UNIV :: 'a set)*

**shows** *infinite (UNIV :: 'a FSet-Lex-Wrapper set)*

$\langle$ *proof* $\rangle$

**lemma** *finite-FSet-Lex-Wrapper-UNIV*:

**assumes** *finite (UNIV :: 'a set)*

**shows** *finite (UNIV :: 'a FSet-Lex-Wrapper set)*  $\langle$ *proof* $\rangle$

**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**  $\langle$ *proof* $\rangle$

**end**

```

instantiation FSet-Lex-Wrapper :: (linorder) cproper-interval begin
fun cproper-interval-FSet-Lex-Wrapper :: 'a FSet-Lex-Wrapper option ⇒ 'a FSet-Lex-Wrapper
option ⇒ bool where
  cproper-interval-FSet-Lex-Wrapper None None ⟷ True
| cproper-interval-FSet-Lex-Wrapper None (Some B) ⟷ (∃ Z. sorted-list-of-fset
(ex Z) < sorted-list-of-fset (ex B))
| cproper-interval-FSet-Lex-Wrapper (Some A) None ⟷ (∃ Z. sorted-list-of-fset
(ex A) < sorted-list-of-fset (ex Z))
| cproper-interval-FSet-Lex-Wrapper (Some A) (Some B) ⟷ (∃ Z. sorted-list-of-fset
(ex A) < sorted-list-of-fset (ex Z) ∧
  sorted-list-of-fset (ex Z) < sorted-list-of-fset (ex B))
declare cproper-interval-FSet-Lex-Wrapper.simps [code del]

lemma lt-of-comp-sorted-list [simp]:
  ID ccompare = Some f ⇒ lt-of-comp f X Z ⟷ sorted-list-of-fset (ex X) <
sorted-list-of-fset (ex Z)
  ⟨proof⟩

instance ⟨proof⟩
end

lemma infinite-term-UNIV[simp, intro]: infinite (UNIV :: ('f,'v)term set)
  ⟨proof⟩

instantiation term :: (type,type) finite-UNIV
begin
definition finite-UNIV = Phantom((('a,'b)term) False)
instance
  ⟨proof⟩
end

instantiation term :: (compare,compare) cproper-interval
begin
definition cproper-interval = (λ ( - :: ('a,'b)term option) - . False)
instance ⟨proof⟩
end

derive (assoclist) mapping-impl FSet-Lex-Wrapper

end
theory Tree-Automata-Impl
imports Tree-Automata-Abstract-Impl
  HOL-Library.List-Lexorder
  HOL-Library.AList-Mapping

```

*Tree-Automata-Class-Instances-Impl*  
*Containers.Containers*

**begin**

**definition** *map-val-of-list* :: ('b ⇒ 'a) ⇒ ('b ⇒ 'c list) ⇒ 'b list ⇒ ('a, 'c list)  
*mapping where*

*map-val-of-list ek ev xs = foldr (λx m. Mapping.update (ek x) (ev x @ case-option  
 Nil id (Mapping.lookup m (ek x))) m) xs Mapping.empty*

**abbreviation** *map-of-list ek ev xs ≡ map-val-of-list ek (λ x. [ev x]) xs*

**lemma** *map-val-of-list-tabulate-conv*:

*map-val-of-list ek ev xs = Mapping.tabulate (sort (remdups (map ek xs))) (λ k.  
 concat (map ev (filter (λ x. k = ek x) xs)))*  
 ⟨*proof*⟩

**lemmas** *map-val-of-list-simp = map-val-of-list-tabulate-conv lookup-tabulate*

## 9.4 Setup for the list implementation of reachable states

**definition** *reach-infer0-cont where*

*reach-infer0-cont Δ =*  
*map r-rhs (filter (λ r. case r of TA-rule f ps p ⇒ ps = []) (sorted-list-of-fset  
 Δ))*

**definition** *reach-infer1-cont* :: ('q :: linorder, 'f :: linorder) *ta-rule fset* ⇒ ('q ×  
 'q) *fset* ⇒ 'q ⇒ 'q *fset* ⇒ 'q *list where*

*reach-infer1-cont Δ Δ<sub>ε</sub> =*  
 (let *rules = sorted-list-of-fset Δ in*  
 let *eps = sorted-list-of-fset Δ<sub>ε</sub> in*  
 let *mapp-r = map-val-of-list fst snd (concat (map (λ r. map (λ q. (q, [r]))  
 (r-lhs-states r)) rules)) in*  
 let *mapp-e = map-of-list fst snd eps in*  
 (λ p bs.  
 (map r-rhs (filter (λ r. case r of TA-rule f qs q ⇒  
 fset-of-list qs |⊆| finsert p bs) (case-option Nil id (Mapping.lookup mapp-r  
 p)))) @  
 case-option Nil id (Mapping.lookup mapp-e p)))

**locale** *reach-rules-fset =*

**fixes** Δ :: ('q :: linorder, 'f :: linorder) *ta-rule fset and* Δ<sub>ε</sub> :: ('q × 'q) *fset*

**begin**

**sublocale** *reach-horn TA Δ Δ<sub>ε</sub> <proof>*

**lemma** *infer1*:

*infer1 p (fset bs) = set (reach-infer1-cont Δ Δ<sub>ε</sub> p bs)*  
 ⟨*proof*⟩

**sublocale** *l*: *horn-fset reach-rules* (*TA*  $\Delta$   $\Delta_\varepsilon$ ) *reach-infer0-cont*  $\Delta$  *reach-infer1-cont*  $\Delta$   $\Delta_\varepsilon$

*<proof>*

**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*  $\Delta$   $\Delta_\varepsilon$  =

*horn-fset-impl.saturate-impl* (*reach-infer0-cont*  $\Delta$ ) (*reach-infer1-cont*  $\Delta$   $\Delta_\varepsilon$ )

**lemma** *reach-fset-impl-sound*:

*reach-cont-impl*  $\Delta$   $\Delta_\varepsilon$  = *Some xs*  $\implies$  *fset xs* = *ta-reach* (*TA*  $\Delta$   $\Delta_\varepsilon$ )

*<proof>*

**lemma** *reach-fset-impl-complete*:

*reach-cont-impl*  $\Delta$   $\Delta_\varepsilon \neq \text{None}$

*<proof>*

**lemma** *reach-impl* [*code*]:

*ta-reachable* (*TA*  $\Delta$   $\Delta_\varepsilon$ ) = *the* (*reach-cont-impl*  $\Delta$   $\Delta_\varepsilon$ )

*<proof>*

## 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*  $\Delta$   $\Delta_\varepsilon$  =

(*let rules* = *sorted-list-of-fset*  $\Delta$  *in*

*let eps* = *sorted-list-of-fset*  $\Delta_\varepsilon$  *in*

*let mapp-r* = *map-of-list* ( $\lambda$  *r*. *r-rhs r*) *r-lhs-states rules in*

*let mapp-e* = *map-of-list snd fst eps in*

( $\lambda$  *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**  $\Delta$  :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* **and**  $\Delta_\varepsilon$  :: (*'q*  $\times$  *'q*) *fset* **and** *P* :: *'q fset*

**begin**

**sublocale** *productive-horn* *TA*  $\Delta$   $\Delta_\varepsilon$  *P* *<proof>*

**lemma** *infer1*:

*infer1 p* (*fset bs*) = *set* (*productive-infer1-cont*  $\Delta$   $\Delta_\varepsilon$  *p bs*)

*<proof>*

**sublocale** *l*: horn-fset productive-rules  $P$  ( $TA \Delta \Delta_\varepsilon$ ) sorted-list-of-fset  $P$  productive-infer1-cont  $\Delta \Delta_\varepsilon$

*<proof>*

**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  $P \Delta \Delta_\varepsilon$  = Some  $xs \implies$  fset  $xs$  = ta-productive-ind  $P$  ( $TA \Delta \Delta_\varepsilon$ )*

*<proof>*

**lemma** *productive-cont-impl-complete*:

*productive-cont-impl  $P \Delta \Delta_\varepsilon \neq$  None*  
*<proof>*

**lemma** *productive-impl [code]*:

*ta-productive  $P$  ( $TA \Delta \Delta_\varepsilon$ ) = the (productive-cont-impl  $P \Delta \Delta_\varepsilon$ )*

*<proof>*

## 9.6 Setup for the implementation of power set construction states

**abbreviation** *r-statesl*  $r \equiv$  *length (r-lhs-states  $r$ )*

**definition** *ps-reachable-states-list* **where**

*ps-reachable-states-list mapp-r mapp-e  $f$   $ps$  =*

*(let  $R$  = filter ( $\lambda r$ . list-all2 ( $|\in|$ ) (r-lhs-states  $r$ )  $ps$ )*

*(case-option Nil id (Mapping.lookup mapp-r ( $f$ , length  $ps$ ))) in*

*let  $S$  = map r-rhs  $R$  in*

*$S$  @ concat (map (case-option Nil id  $\circ$  Mapping.lookup mapp-e)  $S$ ))*

**lemma** *ps-reachable-states-list-sound*:

**assumes** *length  $ps$  =  $n$*

**and** *mapp-r*: *case-option Nil id (Mapping.lookup mapp-r ( $f$ ,  $n$ )) =*

*filter ( $\lambda r$ . r-root  $r$  =  $f \wedge$  r-statesl  $r$  =  $n$ ) (sorted-list-of-fset  $\Delta$ )*

**and** *mapp-e*:  $\bigwedge p$ . *case-option Nil id (Mapping.lookup mapp-e  $p$ ) =*

*map snd (filter ( $\lambda q$ . fst  $q$  =  $p$ ) (sorted-list-of-fset ( $\Delta_\varepsilon|^\dagger|$ )))*

**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>*

**lemma** *rule-target-statesI*:

$\exists r \mid \in \Delta. r\text{-rhs } r = q \implies q \mid \in \text{rule-target-states } \Delta$   
 ⟨proof⟩

**definition** *ps-states-infer0-cont* :: ('q :: linorder, 'f :: linorder) ta-rule fset  $\Rightarrow$

('q  $\times$  'q) fset  $\Rightarrow$  'q FSet-Lex-Wrapper list **where**

*ps-states-infer0-cont*  $\Delta \Delta_\varepsilon =$

(let sig = filter ( $\lambda r. r\text{-lhs-states } r = []$ ) (sorted-list-of-fset  $\Delta$ ) in

filter ( $\lambda p. \text{ex } p \neq \{|\}$ ) (map ( $\lambda r. \text{Wrapp } (\text{ps-reachable-states } (TA \Delta \Delta_\varepsilon)$   
 (r-root r) [])) sig))

**definition** *ps-states-infer1-cont* :: ('q :: linorder, 'f :: linorder) ta-rule fset  $\Rightarrow$  ('q  
 $\times$  'q) fset  $\Rightarrow$

'q FSet-Lex-Wrapper  $\Rightarrow$  'q FSet-Lex-Wrapper fset  $\Rightarrow$  'q FSet-Lex-Wrapper list

**where**

*ps-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-list-of-fset  $\Delta$ ))) in

let arities = remdups (map snd sig) in

let etr = sorted-list-of-fset ( $\Delta_\varepsilon|^{+}$ ) in

let mapp-r = map-of-list ( $\lambda r. (r\text{-root } r, r\text{-statesl } r)$ ) id (sorted-list-of-fset  $\Delta$ )

in

let mapp-e = map-of-list fst snd etr in

( $\lambda p \text{ bs.}$

(let states = sorted-list-of-fset (finsert p bs) in

let arity-to-states-map = Mapping.tabulate arities ( $\lambda n. \text{list-of-permutation-element-}n$   
 p n states) in

let res = map ( $\lambda (f, n).$

map ( $\lambda s. \text{let rules} = \text{the } (\text{Mapping.lookup } mapp\text{-}r \text{ } (f, n)) \text{ in}$

Wrapp (fset-of-list (ps-reachable-states-list mapp-r mapp-e f (map ex s))))

(the (Mapping.lookup arity-to-states-map n))))

sig in

filter ( $\lambda p. \text{ex } p \neq \{|\}$ ) (concat res))))

**locale** *ps-states-fset* =

**fixes**  $\Delta :: ('q :: \text{linorder}, 'f :: \text{linorder}) \text{ta-rule fset}$  **and**  $\Delta_\varepsilon :: ('q \times 'q) \text{fset}$

**begin**

**sublocale** *ps-states-horn* TA  $\Delta \Delta_\varepsilon$  ⟨proof⟩

**lemma** *infer0*: *infer0* = set (ps-states-infer0-cont  $\Delta \Delta_\varepsilon$ )

⟨proof⟩

**lemma** *r-lhs-states-nConst*:

$r\text{-lhs-states } r \neq [] \implies r\text{-statesl } r \neq 0$  **for**  $r$  ⟨proof⟩

**lemma** *filter-empty-conv'*:

$[] = \text{filter } P \text{ } xs \iff (\forall x \in \text{set } xs. \neg P x)$

*<proof>*

**lemma** *infer1*:

*infer1*  $p$  (fset  $bs$ ) = set (ps-states-infer1-cont  $\Delta$   $\Delta_\epsilon$   $p$   $bs$ ) (is ? $Ls$  = ? $Rs$ )  
*<proof>*

**sublocale**  $l$ : horn-fset ps-states-rules (TA  $\Delta$   $\Delta_\epsilon$ ) ps-states-infer0-cont  $\Delta$   $\Delta_\epsilon$  ps-states-infer1-cont  
 $\Delta$   $\Delta_\epsilon$

*<proof>*

**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*  $\Delta$   $\Delta_\epsilon$  =

$horn-fset-impl.saturate-impl$  (ps-states-infer0-cont  $\Delta$   $\Delta_\epsilon$ ) (ps-states-infer1-cont  
 $\Delta$   $\Delta_\epsilon$ )

**lemma** *ps-states-fset-impl-sound*:

**assumes** *ps-states-fset-impl*  $\Delta$   $\Delta_\epsilon$  = *Some*  $xs$

**shows**  $xs$  = ps-states (TA  $\Delta$   $\Delta_\epsilon$ )

*<proof>*

**lemma** *ps-states-fset-impl-complete*:

*ps-states-fset-impl*  $\Delta$   $\Delta_\epsilon$   $\neq$  *None*

*<proof>*

**lemma** *ps-ta-impl* [code]:

*ps-ta* (TA  $\Delta$   $\Delta_\epsilon$ ) =

(let  $xs$  = the (ps-states-fset-impl  $\Delta$   $\Delta_\epsilon$ ) in

TA (ps-rules (TA  $\Delta$   $\Delta_\epsilon$ )  $xs$ ) {||})

*<proof>*

**lemma** *ps-reg-impl* [code]:

*ps-reg* (Reg  $Q$  (TA  $\Delta$   $\Delta_\epsilon$ )) =

(let  $xs$  = the (ps-states-fset-impl  $\Delta$   $\Delta_\epsilon$ ) in

Reg (ffilter ( $\lambda S. Q \mid\cap\mid$  ex  $S \neq \{\mid\mid\}$ )  $xs$ )

(TA (ps-rules (TA  $\Delta$   $\Delta_\epsilon$ )  $xs$ ) {||}))

*<proof>*

**lemma** *prod-ta-zip* [code]:

*prod-ta-rules* ( $\mathcal{A} :: ('q1 :: linorder, 'f :: linorder)$  ta) ( $\mathcal{B} :: ('q2 :: linorder, 'f ::$   
 $linorder)$  ta) =

(let  $sig$  = sorted-list-of-fset (ta-sig  $\mathcal{A} \mid\cap\mid$  ta-sig  $\mathcal{B}$ ) in

let  $mapA$  = map-of-list ( $\lambda r. (r-root$   $r, r-statesl$   $r)$ ) *id* (sorted-list-of-fset (rules

```

A) in
  let mapB = map-of-list (λr. (r-root r, r-statesl r)) id (sorted-list-of-fset (rules
B) in
  let merge = (λ (ra, rb). TA-rule (r-root ra) (zip (r-lhs-states ra) (r-lhs-states
rb)) (r-rhs ra, r-rhs rb)) in
  fset-of-list (
    concat (map (λ (f, n). map merge
      (List.product (the (Mapping.lookup mapA (f, n))) (the (Mapping.lookup
mapB (f, n)))))) sig)))
  (is ?Ls = ?Rs)
⟨proof⟩

```

```

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)
⟨proof⟩

```

```

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
⟨proof⟩

```

```

lemma ta-der-strict-Fun:
  q |∈| ta-der-strict A (Fun f ts) ⟷ (∃ ps p. TA-rule f ps p |∈| (rules A) ∧
  (p = q ∨ (p, q) |∈| (eps A)+) ∧ length ps = length ts ∧
  (∀ i < length ts. ps ! i |∈| ta-der-strict A (ts ! i))) (is ?Ls ⟷ ?Rs)
⟨proof⟩

```

```

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 A t |⊆| ta-der A t
⟨proof⟩

```

**lemma** *ta-der-strict-ta-der-eq-on-ground*:

**assumes** *ground t*

**shows** *ta-der A t = ta-der-strict A t*

*<proof>*

**lemma** *ta-der-to-ta-strict*:

**assumes**  $q \in | \text{ta-der } A \ C \langle \text{Var } p \rangle$  **and** *ground-ctxt C*

**shows**  $\exists q'. (p = q' \vee (p, q') \in | \text{eps } A |^+) \wedge q \in | \text{ta-der-strict } A \ C \langle \text{Var } q' \rangle$

*<proof>*

**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*

*<proof>*

**inductive-set** *Q-inf for A where*

*trans: (p, q)  $\in$  Q-inf A  $\implies$  (q, r)  $\in$  Q-inf A  $\implies$  (p, r)  $\in$  Q-inf A*

| *rule: (None, Some f) qs  $\rightarrow$  q  $\in$  | rules A  $\implies$  i < length qs  $\implies$  (qs ! i, q)  $\in$  Q-inf A*

| *eps: (p, q)  $\in$  Q-inf A  $\implies$  (q, r)  $\in$  | eps A  $\implies$  (p, r)  $\in$  Q-inf A*

**abbreviation** *Q-inf-e A  $\equiv$  {q | p q. (p, p)  $\in$  Q-inf A  $\wedge$  (p, q)  $\in$  Q-inf A}*

**lemma** *Q-inf-states-ta-states*:

**assumes**  $(p, q) \in \text{Q-inf } A$

**shows**  $p \in | \text{Q } A \ q \in | \text{Q } A$

*<proof>*

**lemma** *Q-inf-finite*:

*finite (Q-inf A) finite (Q-inf-e A)*

*<proof>*

**context**

**includes** *fset.lifting*

**begin**

**lift-definition** *fQ-inf :: ('a, 'b option  $\times$  'c option) ta  $\Rightarrow$  ('a  $\times$  'a) fset is Q-inf*

*<proof>*

**lift-definition** *fQ-inf-e :: ('a, 'b option  $\times$  'c option) ta  $\Rightarrow$  'a fset is Q-inf-e*

*<proof>*

**end**

**lemma** *Q-inf-ta-eps-Q-inf*:

**assumes**  $(p, q) \in Q\text{-inf } \mathcal{A}$  **and**  $(q, q') \in (eps \ \mathcal{A})^+|$

**shows**  $(p, q') \in Q\text{-inf } \mathcal{A}$  *<proof>*

**lemma** *lhs-state-rule*:

**assumes**  $(p, q) \in Q\text{-inf } \mathcal{A}$

**shows**  $\exists f \ q \ r. (None, Some \ f) \ q \rightarrow r \in rules \ \mathcal{A} \wedge p \in fset\text{-of-list } q \mathcal{A}$

*<proof>*

**lemma** *Q-inf-reach-state-rule*:

**assumes**  $(p, q) \in Q\text{-inf } \mathcal{A}$  **and**  $Q \ \mathcal{A} \sqsubseteq ta\text{-reachable } \mathcal{A}$

**shows**  $\exists ss \ ts \ f \ C. q \in ta\text{-der } \mathcal{A} (More \ (None, Some \ f) \ ss \ C \ ts) (Var \ p) \wedge$   
*ground-ctxt*  $(More \ (None, Some \ f) \ ss \ C \ ts)$

**(is**  $\exists ss \ ts \ f \ C. ?P \ ss \ ts \ f \ C \ q \ p)$

*<proof>*

**lemma** *rule-target-Q-inf*:

**assumes**  $(None, Some \ f) \ q \rightarrow q' \in rules \ \mathcal{A}$  **and**  $i < length \ q \mathcal{A}$

**shows**  $(q \ ! \ i, q') \in Q\text{-inf } \mathcal{A}$  *<proof>*

**lemma** *rule-target-eps-Q-inf*:

**assumes**  $(None, Some \ f) \ q \rightarrow q' \in (eps \ \mathcal{A})^+|$

**and**  $i < length \ q \mathcal{A}$

**shows**  $(q \ ! \ i, q) \in Q\text{-inf } \mathcal{A}$

*<proof>*

**lemma** *step-in-Q-inf*:

**assumes**  $q \in ta\text{-der-strict } \mathcal{A} (map\text{-funs-term } (\lambda f. (None, Some \ f)) (Fun \ f \ (ss$   
 $@ \ Var \ p \ \# \ ts)))$

**shows**  $(p, q) \in Q\text{-inf } \mathcal{A}$

*<proof>*

**lemma** *ta-der-Q-inf*:

**assumes**  $q \in ta\text{-der-strict } \mathcal{A} (map\text{-funs-term } (\lambda f. (None, Some \ f)) (C \ (Var \ p)))$   
**and**  $C \neq Hole$

**shows**  $(p, q) \in Q\text{-inf } \mathcal{A}$  *<proof>*

**lemma** *Q-inf-e-infinite-terms-res*:

**assumes**  $q \in Q\text{-inf-e } \mathcal{A}$  **and**  $Q \ \mathcal{A} \sqsubseteq ta\text{-reachable } \mathcal{A}$

**shows**  $infinite \ \{t. q \in ta\text{-der } \mathcal{A} (term\text{-of-gterm } t) \wedge fst \ (groot\text{-sym } t) = None\}$   
*<proof>*

**lemma** *gfun-at-after-hole-pos*:  
**assumes** *ghole-pos*  $C \leq_p p$   
**shows** *gfun-at*  $C \langle t \rangle_G p = \text{gfun-at } t (p \text{ } _p \text{ } \text{ghole-pos } C)$   $\langle \text{proof} \rangle$

**lemma** *pos-diff-0 [simp]*:  $p \text{ } _p \text{ } p = []$   
 $\langle \text{proof} \rangle$

**lemma** *Max-suffI*:  $\text{finite } A \implies A = B \implies \text{Max } A = \text{Max } B$   
 $\langle \text{proof} \rangle$

**lemma** *nth-args-depth-eqI*:  
**assumes**  $\text{length } ss = \text{length } ts$   
**and**  $\bigwedge i. i < \text{length } ts \implies \text{depth } (ss ! i) = \text{depth } (ts ! i)$   
**shows**  $\text{depth } (\text{Fun } f \text{ } ss) = \text{depth } (\text{Fun } g \text{ } ts)$   
 $\langle \text{proof} \rangle$

**lemma** *prod-automata-from-none-root-dec*:  
**assumes** *gta-lang*  $Q \mathcal{A} \subseteq \{\text{gpair } s \ t \mid s \ t. \text{funas-gterm } s \subseteq \mathcal{F} \wedge \text{funas-gterm } t \subseteq \mathcal{F}\}$   
**and**  $q \in | \text{ta-der } \mathcal{A} \text{ (term-of-gterm } t) \text{ and fst (groot-sym } t) = \text{None}$   
**and**  $Q \mathcal{A} \subseteq | \text{ta-reachable } \mathcal{A} \text{ and } q \in | \text{ta-productive } Q \mathcal{A}$   
**shows**  $\exists u. t = \text{gterm-to-None-Some } u \wedge \text{funas-gterm } u \subseteq \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *infinite-set-dec-infinite*:  
**assumes** *infinite*  $S$  **and**  $\bigwedge s. s \in S \implies \exists t. f \ t = s \wedge P \ t$   
**shows**  $\text{infinite } \{t \mid t \ s. s \in S \wedge f \ t = s \wedge P \ t\}$  **(is infinite ?T)**  
 $\langle \text{proof} \rangle$

**lemma** *Q-inf-exec-impl-Q-inf*:  
**assumes** *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**  $Q \mathcal{A} \subseteq | \text{ta-reachable } \mathcal{A} \text{ and } Q \mathcal{A} \subseteq | \text{ta-productive } Q \mathcal{A}$   
**and**  $q \in Q\text{-inf-e } \mathcal{A}$   
**shows**  $q \in | Q\text{-infy } \mathcal{A} \ \mathcal{F}$   
 $\langle \text{proof} \rangle$

**lemma** *Q-inf-impl-Q-inf-exec*:  
**assumes**  $q \in | Q\text{-infy } \mathcal{A} \ \mathcal{F}$   
**shows**  $q \in Q\text{-inf-e } \mathcal{A}$   
 $\langle \text{proof} \rangle$

**lemma** *Q-infty-fQ-inf-e-conv*:

**assumes** *gta-lang*  $Q \mathcal{A} \subseteq \{gpair \ s \ t \mid s \ t. \text{funas-gterm } s \subseteq \text{fset } \mathcal{F} \wedge \text{funas-gterm } t \subseteq \text{fset } \mathcal{F}\}$

**and**  $Q \mathcal{A} \mid \subseteq \mid \text{ta-reachable } \mathcal{A}$  **and**  $Q \mathcal{A} \mid \subseteq \mid \text{ta-productive } Q \mathcal{A}$

**shows** *Q-infty*  $\mathcal{A} \mathcal{F} = \text{fQ-inf-e } \mathcal{A}$

*<proof>*

**definition** *Inf-reg-impl* **where**

*Inf-reg-impl*  $R = \text{Inf-reg } R \ (\text{fQ-inf-e } (\text{ta } R))$

**lemma** *Inf-reg-impl-sound*:

**assumes**  $\mathcal{L} \mathcal{A} \subseteq \{gpair \ s \ t \mid s \ t. \text{funas-gterm } s \subseteq \text{fset } \mathcal{F} \wedge \text{funas-gterm } t \subseteq \text{fset } \mathcal{F}\}$

**and**  $Q_r \mathcal{A} \mid \subseteq \mid \text{ta-reachable } (\text{ta } \mathcal{A})$  **and**  $Q_r \mathcal{A} \mid \subseteq \mid \text{ta-productive } (\text{fin } \mathcal{A}) \ (\text{ta } \mathcal{A})$

**shows**  $\mathcal{L} (\text{Inf-reg-impl } \mathcal{A}) = \mathcal{L} (\text{Inf-reg } \mathcal{A} \ (\text{Q-infty } (\text{ta } \mathcal{A}) \ \mathcal{F}))$

*<proof>*

**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*  $\Delta \ \Delta_E \equiv \text{TA} \ (\text{fset-of-list } \Delta) \ (\text{fset-of-list } \Delta_E)$

## 10 Computing the epsilon transitions for the composition of GTT's

**definition**  $\Delta_\varepsilon$ -rules  $:: ('q, 'f) \text{ta} \Rightarrow ('q, 'f) \text{ta} \Rightarrow ('q \times 'q) \text{horn set}$  **where**

$\Delta_\varepsilon$ -rules  $A \ B =$

$\{zip \ ps \ qs \ \rightarrow_h \ (p, q) \mid f \ ps \ p \ qs \ q. \ f \ ps \ \rightarrow \ p \mid \in \mid \text{rules } A \wedge f \ qs \ \rightarrow \ q \mid \in \mid \text{rules } B$

$\wedge \text{length } ps = \text{length } qs\} \cup$

$\{[(p, q)] \ \rightarrow_h \ (p', q) \mid p \ p' \ q. \ (p, p') \mid \in \mid \text{eps } A\} \cup$

$\{[(p, q)] \ \rightarrow_h \ (p, q') \mid p \ q \ q'. \ (q, q') \mid \in \mid \text{eps } B\}$

**locale**  $\Delta_\varepsilon$ -horn  $=$

**fixes**  $A :: ('q, 'f) \text{ta}$  **and**  $B :: ('q, 'f) \text{ta}$

**begin**

**sublocale** *horn*  $\Delta_\varepsilon$ -rules  $A \ B$  *<proof>*

**lemma**  $\Delta_\varepsilon$ -infer0:

*infer0*  $= \{(p, q) \mid f \ p \ q. \ f \ [] \ \rightarrow \ p \mid \in \mid \text{rules } A \wedge f \ [] \ \rightarrow \ q \mid \in \mid \text{rules } B\}$

*<proof>*

**lemma**  $\Delta_\varepsilon$ -infer1:

$infer1\ pq\ X = \{(p, q) \mid f\ ps\ p\ qs\ q. f\ ps \rightarrow p \mid \in \mid rules\ A \wedge f\ qs \rightarrow q \mid \in \mid rules\ B \wedge length\ ps = length\ qs \wedge (fst\ pq, snd\ pq) \in set\ (zip\ ps\ qs) \wedge set\ (zip\ ps\ qs) \subseteq insert\ pq\ X\} \cup \{(p', snd\ pq) \mid p\ p'. (p, p') \mid \in \mid eps\ A \wedge p = fst\ pq\} \cup \{(fst\ pq, q') \mid q\ q'. (q, q') \mid \in \mid eps\ B \wedge q = snd\ pq\}$   
 $\langle proof \rangle$

**lemma**  $\Delta_\varepsilon$ -sound:

$\Delta_\varepsilon$ -set  $A\ B = saturate$   
 $\langle proof \rangle$

**end**

## 11 Computing the epsilon transitions for the transitive closure of GTT's

**definition**  $\Delta$ -trancl-rules ::  $('q, 'f)\ ta \Rightarrow ('q, 'f)\ ta \Rightarrow ('q \times 'q)\ horn\ set$  **where**

$\Delta$ -trancl-rules  $A\ B =$   
 $\Delta_\varepsilon$ -rules  $A\ B \cup \{[(p, q), (q, r)] \rightarrow_h (p, r) \mid p\ q\ r. True\}$

**locale**  $\Delta$ -trancl-horn =

**fixes**  $A :: ('q, 'f)\ ta$  **and**  $B :: ('q, 'f)\ ta$   
**begin**

**sublocale** *horn*  $\Delta$ -trancl-rules  $A\ B$   $\langle proof \rangle$

**lemma**  $\Delta$ -trancl-infer0:

$infer0 = horn.infer0\ (\Delta_\varepsilon$ -rules  $A\ B)$   
 $\langle proof \rangle$

**lemma**  $\Delta$ -trancl-infer1:

$infer1\ pq\ X = horn.infer1\ (\Delta_\varepsilon$ -rules  $A\ B)\ pq\ X \cup \{(r, snd\ pq) \mid r\ p'. (r, p') \in X \wedge p' = fst\ pq\} \cup \{(fst\ pq, r) \mid q'\ r. (q', r) \in (insert\ pq\ X) \wedge q' = snd\ pq\}$   
 $\langle proof \rangle$

**lemma**  $\Delta$ -trancl-sound:

$\Delta$ -trancl-set  $A\ B = saturate$   
 $\langle proof \rangle$

**end**

## 12 Computing the epsilon transitions for the transitive closure of pair automata

**definition**  $\Delta$ -Atr-rules :: ('q × 'q) fset ⇒ ('q, 'f) ta ⇒ ('q, 'f) ta ⇒ ('q × 'q) horn set **where**

$$\begin{aligned} \Delta\text{-Atr-rules } Q A B = & \\ & \{\ [] \rightarrow_h (p, q) \mid p q. (p, q) \in Q \} \cup \\ & \{ [(p, q), (r, v)] \rightarrow_h (p, v) \mid p q r v. (q, r) \in \Delta_\varepsilon B A \} \end{aligned}$$

**locale**  $\Delta$ -Atr-horn =

**fixes**  $Q :: ('q \times 'q) \text{ fset}$  **and**  $A :: ('q, 'f) \text{ ta}$  **and**  $B :: ('q, 'f) \text{ ta}$   
**begin**

**sublocale** horn  $\Delta$ -Atr-rules  $Q A B$   $\langle \text{proof} \rangle$

**lemma**  $\Delta$ -Atr-infer0: infer0 = fset  $Q$   
 $\langle \text{proof} \rangle$

**lemma**  $\Delta$ -Atr-infer1:

$$\begin{aligned} \text{infer1 } pq X = & \{ (p, \text{snd } pq) \mid p q. (p, q) \in X \wedge (q, \text{fst } pq) \in \Delta_\varepsilon B A \} \cup \\ & \{ (\text{fst } pq, v) \mid q r v. (\text{snd } pq, r) \in \Delta_\varepsilon B A \wedge (r, v) \in X \} \cup \\ & \{ (\text{fst } pq, \text{snd } pq) \mid q. (\text{snd } pq, \text{fst } pq) \in \Delta_\varepsilon B A \} \end{aligned}$$

$\langle \text{proof} \rangle$

**lemma**  $\Delta$ -Atr-sound:

$$\Delta\text{-Atrans-set } Q A B = \text{saturate}$$

$\langle \text{proof} \rangle$

**end**

## 13 Computing the Q infinity set for the infinity predicate automaton

**definition**  $Q$ -inf-rules :: ('q, 'f option × 'g option) ta ⇒ ('q × 'q) horn set **where**

$$\begin{aligned} Q\text{-inf-rules } A = & \\ & \{ [] \rightarrow_h (ps ! i, p) \mid f ps p i. (None, Some f) ps \rightarrow p \in \text{rules } A \wedge i < \text{length } ps \} \\ \cup & \\ & \{ [(p, q)] \rightarrow_h (p, r) \mid p q r. (q, r) \in \text{eps } A \} \cup \\ & \{ [(p, q), (q, r)] \rightarrow_h (p, r) \mid p q r. \text{True} \} \end{aligned}$$

**locale**  $Q$ -horn =

**fixes**  $A :: ('q, 'f \text{ option} \times 'g \text{ option}) \text{ ta}$   
**begin**

**sublocale** horn  $Q$ -inf-rules  $A$   $\langle \text{proof} \rangle$

**lemma**  $Q$ -infer0:

$$\text{infer0} = \{ (ps ! i, p) \mid f ps p i. (None, Some f) ps \rightarrow p \in \text{rules } A \wedge i < \text{length}$$

$ps\}$   
 $\langle proof \rangle$

**lemma** *Q-infer1*:

$infer1\ pq\ X = \{(fst\ pq,\ r) \mid q\ r.\ (q,\ r) \in |eps\ A \wedge q = snd\ pq\} \cup$   
 $\{(r,\ snd\ pq) \mid r\ p'.\ (r,\ p') \in X \wedge p' = fst\ pq\} \cup$   
 $\{(fst\ pq,\ r) \mid q'\ r.\ (q',\ r) \in (insert\ pq\ X) \wedge q' = snd\ pq\}$   
 $\langle proof \rangle$

**lemma** *Q-sound*:

$Q-inf\ A = saturate$   
 $\langle proof \rangle$

**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**  $\Delta_\varepsilon$ -*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**  $\Delta_\varepsilon$ -*infer1-cont* **where**

$\Delta_\varepsilon$ -*infer1-cont*  $\Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon} =$   
 $(let\ (arules,\ aeeps) = (sorted-list-of-fset\ \Delta_A,\ sorted-list-of-fset\ \Delta_{A\varepsilon})\ in$   
 $let\ (brules,\ beeps) = (sorted-list-of-fset\ \Delta_B,\ sorted-list-of-fset\ \Delta_{B\varepsilon})\ in$   
 $let\ prules = List.product\ arules\ brules\ in$   
 $(\lambda\ pq\ bs.$   
 $map\ (map-prod\ r-rhs\ r-rhs)\ (filter\ (\lambda(ra,\ rb).\ case\ (ra,\ rb)\ of\ (TA-rule\ f\ ps\ p,$   
 $TA-rule\ g\ qs\ q) \Rightarrow$   
 $f = g \wedge length\ ps = length\ qs \wedge (fst\ pq,\ snd\ pq) \in set\ (zip\ ps\ qs) \wedge$   
 $set\ (zip\ ps\ qs) \subseteq insert\ (fst\ pq,\ snd\ pq)\ (fset\ bs))\ prules) @$   
 $map\ (\lambda(p,\ p').\ (p',\ snd\ pq))\ (filter\ (\lambda(p,\ p') \Rightarrow p = fst\ pq)\ aeeps) @$   
 $map\ (\lambda(q,\ q').\ (fst\ pq,\ q'))\ (filter\ (\lambda(q,\ q') \Rightarrow q = snd\ pq)\ beeps)))$

**locale**  $\Delta_\varepsilon$ -*fset* =

**fixes**  $\Delta_A :: ('q :: \text{linorder}, 'f :: \text{linorder}) \text{ ta-rule fset}$  **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**  $\Delta_\varepsilon\text{-horn}$   $A B$   $\langle \text{proof} \rangle$

**sublocale**  $l$ :  $\text{horn-fset } \Delta_\varepsilon\text{-rules } A B \Delta_\varepsilon\text{-infer0-cont } \Delta_A \Delta_B \Delta_\varepsilon\text{-infer1-cont } \Delta_A$   
 $\Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{infer} = l.\text{infer0 } l.\text{infer1}$   
**lemmas**  $\text{saturate-impl-sound} = l.\text{saturate-impl-sound}$   
**lemmas**  $\text{saturate-impl-complete} = l.\text{saturate-impl-complete}$

**end**

**definition**  $\Delta_\varepsilon\text{-impl}$  **where**  
 $\Delta_\varepsilon\text{-impl } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} = \text{horn-fset-impl.saturate-impl } (\Delta_\varepsilon\text{-infer0-cont } \Delta_A$   
 $\Delta_B) (\Delta_\varepsilon\text{-infer1-cont } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon})$

**lemma**  $\Delta_\varepsilon\text{-impl-sound}$ :  
**assumes**  $\Delta_\varepsilon\text{-impl } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} = \text{Some } xs$   
**shows**  $xs = \Delta_\varepsilon (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon})$   
 $\langle \text{proof} \rangle$

**lemma**  $\Delta_\varepsilon\text{-impl-complete}$ :  
**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**  $\Delta_\varepsilon\text{-impl } \Delta_A \Delta_{\varepsilon A} \Delta_B \Delta_{\varepsilon B} \neq \text{None}$   $\langle \text{proof} \rangle$

**lemma**  $\Delta_\varepsilon\text{-impl}$   $[\text{code}]$ :  
 $\Delta_\varepsilon (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon}) = \text{the } (\Delta_\varepsilon\text{-impl } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon})$   
 $\langle \text{proof} \rangle$

## 15 Computing the epsilon transitions for the transitive closure of GTT's

**definition**  $\Delta\text{-trancl-infer0}$  **where**  
 $\Delta\text{-trancl-infer0 } \Delta_A \Delta_B = \Delta_\varepsilon\text{-infer0-cont } \Delta_A \Delta_B$

**definition**  $\Delta\text{-trancl-infer1}$   $:: ('q :: \text{linorder}, 'f :: \text{linorder}) \text{ ta-rule fset} \Rightarrow ('q \times 'q) \text{ fset}$   
 $\Rightarrow ('q, 'f) \text{ ta-rule fset} \Rightarrow ('q \times 'q) \text{ fset}$   
 $\Rightarrow 'q \times 'q \Rightarrow ('q \times 'q) \text{ fset} \Rightarrow ('q \times 'q) \text{ list}$  **where**  
 $\Delta\text{-trancl-infer1 } \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} pq bs =$

```

Δε-infer1-cont ΔA ΔAε ΔB ΔBε pq bs @
sorted-list-of-fset (
  (λ(r, p'). (r, snd pq)) |'| (ffilter (λ(r, p') ⇒ p' = fst pq) bs) |∪|
  (λ(q', r). (fst pq, r)) |'| (ffilter (λ(q', r) ⇒ q' = snd pq) (finsert pq bs)))

locale Δ-trancl-list =
  fixes ΔA :: ('q :: linorder, 'f :: linorder) ta-rule fset and ΔAε :: ('q × 'q) fset
  and ΔB :: ('q, 'f) ta-rule fset and ΔBε :: ('q × 'q) fset
begin

abbreviation A where A ≡ TA ΔA ΔAε
abbreviation B where B ≡ TA ΔB ΔBε

sublocale Δ-trancl-horn A B ⟨proof⟩

sublocale l: horn-fset Δ-trancl-rules A B
  Δ-trancl-infer0 ΔA ΔB Δ-trancl-infer1 ΔA ΔAε ΔB ΔBε
  ⟨proof⟩

lemmas saturate-impl-sound = l.saturate-impl-sound
lemmas saturate-impl-complete = l.saturate-impl-complete

end

definition Δ-trancl-impl ΔA ΔAε ΔB ΔBε =
  horn-fset-impl.saturate-impl (Δ-trancl-infer0 ΔA ΔB) (Δ-trancl-infer1 ΔA ΔAε
  ΔB ΔBε)

lemma Δ-trancl-impl-sound:
  assumes Δ-trancl-impl ΔA ΔAε ΔB ΔBε = Some xs
  shows xs = Δ-trancl (TA ΔA ΔAε) (TA ΔB ΔBε)
  ⟨proof⟩

lemma Δ-trancl-impl-complete:
  fixes ΔA :: ('q :: linorder, 'f :: linorder) ta-rule fset and ΔB :: ('q, 'f) ta-rule
  fset
  and ΔAε :: ('q × 'q) fset and ΔBε :: ('q × 'q) fset
  shows Δ-trancl-impl ΔA ΔAε ΔB ΔBε ≠ None
  ⟨proof⟩

lemma Δ-trancl-impl [code]:
  Δ-trancl (TA ΔA ΔAε) (TA ΔB ΔBε) = (the (Δ-trancl-impl ΔA ΔAε ΔB ΔBε))
  ⟨proof⟩

```

## 16 Computing the epsilon transitions for the transitive closure of pair automata

**definition**  $\Delta\text{-Atr-infer1-cont} :: ('q :: \text{linorder} \times 'q) \text{fset} \Rightarrow ('q, 'f :: \text{linorder}) \text{ta-rule fset} \Rightarrow ('q \times 'q) \text{fset} \Rightarrow$

$('q, 'f) \text{ta-rule fset} \Rightarrow ('q \times 'q) \text{fset} \Rightarrow 'q \times 'q \Rightarrow ('q \times 'q) \text{fset} \Rightarrow ('q \times 'q) \text{list}$

**where**

$\Delta\text{-Atr-infer1-cont } Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon} =$

$(\text{let } G = \text{sorted-list-of-fset } (\text{the } (\Delta_\epsilon\text{-impl } \Delta_B \Delta_{B\epsilon} \Delta_A \Delta_{A\epsilon})) \text{ in}$

$(\lambda pq \text{ bs.}$

$(\text{let } \text{bs-list} = \text{sorted-list-of-fset } \text{bs} \text{ in}$

$\text{map } (\lambda (p, q). (\text{fst } p, \text{snd } pq)) (\text{filter } (\lambda (p, q). \text{snd } p = \text{fst } q \wedge \text{snd } q = \text{fst } pq) (\text{List.product } \text{bs-list } G)) @$

$\text{map } (\lambda (p, q). (\text{fst } pq, \text{snd } q)) (\text{filter } (\lambda (p, q). \text{snd } p = \text{fst } q \wedge \text{fst } p = \text{snd } pq) (\text{List.product } G \text{ bs-list})) @$

$\text{map } (\lambda (p, q). (\text{fst } pq, \text{snd } pq)) (\text{filter } (\lambda (p, q). \text{snd } pq = p \wedge \text{fst } pq = q) G))))$

**locale**  $\Delta\text{-Atr-fset} =$

**fixes**  $Q :: ('q :: \text{linorder} \times 'q) \text{fset}$  **and**  $\Delta_A :: ('q, 'f :: \text{linorder}) \text{ta-rule fset}$  **and**  $\Delta_{A\epsilon} :: ('q \times 'q) \text{fset}$

**and**  $\Delta_B :: ('q, 'f) \text{ta-rule fset}$  **and**  $\Delta_{B\epsilon} :: ('q \times 'q) \text{fset}$

**begin**

**abbreviation**  $A$  **where**  $A \equiv TA \Delta_A \Delta_{A\epsilon}$

**abbreviation**  $B$  **where**  $B \equiv TA \Delta_B \Delta_{B\epsilon}$

**sublocale**  $\Delta\text{-Atr-horn } Q A B \langle \text{proof} \rangle$

**lemma**  $\text{infer1}$ :

$\text{infer1 } pq (\text{fset } \text{bs}) = \text{set } (\Delta\text{-Atr-infer1-cont } Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon} pq \text{ bs})$

$\langle \text{proof} \rangle$

**sublocale**  $l$ :  $\text{horn-fset } \Delta\text{-Atr-rules } Q A B \text{sorted-list-of-fset } Q \Delta\text{-Atr-infer1-cont}$

$Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon}$

$\langle \text{proof} \rangle$

**lemmas**  $\text{infer} = l.\text{infer0 } l.\text{infer1}$

**lemmas**  $\text{saturate-impl-sound} = l.\text{saturate-impl-sound}$

**lemmas**  $\text{saturate-impl-complete} = l.\text{saturate-impl-complete}$

**end**

**definition**  $\Delta\text{-Atr-impl } Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon} =$

$\text{horn-fset-impl.saturate-impl } (\text{sorted-list-of-fset } Q) (\Delta\text{-Atr-infer1-cont } Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon})$

**lemma**  $\Delta\text{-Atr-impl-sound}$ :

**assumes**  $\Delta\text{-Atr-impl } Q \Delta_A \Delta_{A\epsilon} \Delta_B \Delta_{B\epsilon} = \text{Some } xs$

**shows**  $xs = \Delta\text{-Atrans } Q (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon})$   
 ⟨proof⟩

**lemma**  $\Delta\text{-Atr-impl-complete}$ :

**shows**  $\Delta\text{-Atr-impl } Q \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon} \neq \text{None}$  ⟨proof⟩

**lemma**  $\Delta\text{-Atr-impl}$  [code]:

$\Delta\text{-Atrans } Q (TA \Delta_A \Delta_{A\varepsilon}) (TA \Delta_B \Delta_{B\varepsilon}) = (\text{the } (\Delta\text{-Atr-impl } Q \Delta_A \Delta_{A\varepsilon} \Delta_B \Delta_{B\varepsilon}))$   
 ⟨proof⟩

## 17 Computing the Q infinity set for the infinity predicate automaton

**definition**  $Q\text{-infer0-cont} :: ('q :: \text{linorder}, 'f :: \text{linorder option} \times 'g :: \text{linorder option}) \text{ta-rule fset} \Rightarrow ('q \times 'g) \text{list}$  **where**

$Q\text{-infer0-cont } \Delta = \text{concat } (\text{sorted-list-of-fset } ($   
 $(\lambda r. \text{case } r \text{ of } \text{TA-rule } f \text{ ps } p \Rightarrow \text{map } (\lambda x. \text{Pair } x \text{ } p) \text{ ps}) \mid$   
 $(\text{ffilter } (\lambda r. \text{case } r \text{ of } \text{TA-rule } f \text{ ps } p \Rightarrow \text{fst } f = \text{None} \wedge \text{snd } f \neq \text{None} \wedge \text{ps} \neq [])) \Delta))$

**definition**  $Q\text{-infer1-cont} :: ('q :: \text{linorder} \times 'g) \text{fset} \Rightarrow 'q \times 'g \Rightarrow ('q \times 'g) \text{fset} \Rightarrow ('q \times 'g) \text{list}$  **where**

$Q\text{-infer1-cont } \Delta\varepsilon =$   
 $(\text{let } \text{eps} = \text{sorted-list-of-fset } \Delta\varepsilon \text{ in}$   
 $(\lambda \text{pq } \text{bs}.$   
 $\text{let } \text{bs-list} = \text{sorted-list-of-fset } \text{bs} \text{ in}$   
 $\text{map } (\lambda (q, r). (\text{fst } \text{pq}, r)) (\text{filter } (\lambda (q, r) \Rightarrow q = \text{snd } \text{pq}) \text{eps}) \text{@}$   
 $\text{map } (\lambda (r, p'). (r, \text{snd } \text{pq})) (\text{filter } (\lambda (r, p') \Rightarrow p' = \text{fst } \text{pq}) \text{bs-list}) \text{@}$   
 $\text{map } (\lambda (q', r). (\text{fst } \text{pq}, r)) (\text{filter } (\lambda (q', r) \Rightarrow q' = \text{snd } \text{pq}) (\text{pq} \# \text{bs-list}))))$

**locale**  $Q\text{-fset} =$

**fixes**  $\Delta :: ('q :: \text{linorder}, 'f :: \text{linorder option} \times 'g :: \text{linorder option}) \text{ta-rule fset}$   
**and**  $\Delta\varepsilon :: ('q \times 'g) \text{fset}$   
**begin**

**abbreviation**  $A$  **where**  $A \equiv TA \Delta \Delta\varepsilon$

**sublocale**  $Q\text{-horn } A$  ⟨proof⟩

**sublocale**  $l$ :  $\text{horn-fset } Q\text{-inf-rules } A \text{ } Q\text{-infer0-cont } \Delta \text{ } Q\text{-infer1-cont } \Delta\varepsilon$   
 ⟨proof⟩

**lemmas**  $\text{saturate-impl-sound} = l.\text{saturate-impl-sound}$

**lemmas**  $\text{saturate-impl-complete} = l.\text{saturate-impl-complete}$

**end**

**definition**  $Q\text{-impl}$  **where**

$Q\text{-impl } \Delta \Delta\varepsilon = \text{horn-fset-impl.saturate-impl } (Q\text{-infer0-cont } \Delta) (Q\text{-infer1-cont } \Delta\varepsilon)$

**lemma** *Q-impl-sound*:

$Q\text{-impl } \Delta \Delta\varepsilon = \text{Some } xs \implies \text{fset } xs = Q\text{-inf } (TA \Delta \Delta\varepsilon)$   
 $\langle \text{proof} \rangle$

**lemma** *Q-impl-complete*:

$Q\text{-impl } \Delta \Delta\varepsilon \neq \text{None}$   
 $\langle \text{proof} \rangle$

**definition** *Q-infinity-impl*  $\Delta \Delta\varepsilon = (\text{let } Q = \text{the } (Q\text{-impl } \Delta \Delta\varepsilon) \text{ in}$   
 $\text{snd } |q| ((\text{ffilter } (\lambda (p, q). p = q) Q) |O| Q))$

**lemma** *Q-infinity-impl-fmember*:

$q \in |Q\text{-infinity-impl } \Delta \Delta\varepsilon \iff (\exists p. (p, p) \in | \text{the } (Q\text{-impl } \Delta \Delta\varepsilon) \wedge$   
 $(p, q) \in | \text{the } (Q\text{-impl } \Delta \Delta\varepsilon))$   
 $\langle \text{proof} \rangle$

**lemma** *loop-sound-correct [simp]*:

$\text{fset } (Q\text{-infinity-impl } \Delta \Delta\varepsilon) = Q\text{-inf-e } (TA \Delta \Delta\varepsilon)$   
 $\langle \text{proof} \rangle$

**lemma** *fQ-inf-e-code [code]*:

$\text{fQ-inf-e } (TA \Delta \Delta\varepsilon) = Q\text{-infinity-impl } \Delta \Delta\varepsilon$   
 $\langle \text{proof} \rangle$

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.