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

Alexander Lochmann   Bertram Felgenhauer
Christian Sternagel   René Thiemann   Thomas Sternagel

May 6, 2022

Abstract

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

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

Contents

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

1 Introduction

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

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

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

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

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

2 Preliminaries

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

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

2.1.1 Positions

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

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

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

definition *position-less-eq* (**infixl** $\leq_p$ *67*) **where**
$p \leq_p q \longleftrightarrow (\exists\ r.\ p\ @\ r = q)$

abbreviation *position-less* (**infixl** $<_p$ *67*) **where**
$p <_p q \equiv p \neq q \wedge p \leq_p q$

definition *position-par* (**infixl** ⊥ *67*) **where**
$p \perp q \longleftrightarrow \neg\ (p \leq_p q) \wedge \neg\ (q \leq_p p)$

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

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

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

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

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

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

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

2.1.2 Computing the signature

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

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

2.1.3 Groundness

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

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

2.1.4 Depth

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

2.1.5 Type conversion

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

abbreviation *map-vars-term f* $\equiv$ *map-term* (λ *x*. *x*) *f*
abbreviation *map-funs-term f* $\equiv$ *map-term f* (λ *x*. *x*)
abbreviation *map-both f* $\equiv$ *map-prod f f*

definition *adapt-vars* :: ('*f*, '*q*) *term* $\Rightarrow$ ('*f*,'*v*)*term* **where**
[*code del*]: *adapt-vars* $\equiv$ *map-vars-term* (λ-. *undefined*)

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

2.2 Properties of *pos*

lemma *position-less-eq-induct* [*consumes 1*]:
assumes $p \leq_p q$ **and** $\bigwedge p.\ P\ p\ p$
and $\bigwedge p\ q\ r.\ p \leq_p q \Longrightarrow P\ p\ q \Longrightarrow P\ p\ (q\ @\ r)$
shows $P\ p\ q$ ⟨*proof*⟩

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

lemma *remove-prefix-Nil* [*simp*]:
remove-prefix xs xs = *Some* []
⟨*proof*⟩

lemma *remove-prefix-Some*:
assumes *remove-prefix xs ys* = *Some zs*
shows *ys* = *xs* @ *zs* ⟨*proof*⟩

lemma *remove-prefix-append*:
remove-prefix xs (*xs* @ *ys*) = *Some ys*
⟨*proof*⟩

lemma *remove-prefix-iff*:
remove-prefix xs ys = *Some zs* $\longleftrightarrow$ *ys* = *xs* @ *zs*
⟨*proof*⟩

lemma *position-less-eq-remove-prefix*:
$p \leq_p q \Longrightarrow$ *remove-prefix p q* $\neq$ *None*
⟨*proof*⟩

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

lemma *position-less-refl* [*simp*]: $p \leq_p p$
⟨*proof*⟩

lemma *position-less-eq-Cons* [*simp*]:
$(i\ \#\ ps) \leq_p (j\ \#\ qs) \longleftrightarrow i = j \land ps \leq_p qs$
⟨*proof*⟩

lemma *position-less-Nil-is-bot* [*simp*]: $[] \leq_p p$
⟨*proof*⟩

lemma *position-less-Nil-is-bot2* [*simp*]: $p \leq_p [] \longleftrightarrow p = []$
⟨*proof*⟩

lemma *position-diff-Nil* [*simp*]: $q -_p [] = q$
⟨*proof*⟩

lemma *position-diff-Cons* [*simp*]:
$(i \# ps) -_p (i \# qs) = ps -_p qs$
⟨*proof*⟩

lemma *Nil-not-par* [*simp*]:
$[] \perp p \longleftrightarrow False$
$p \perp [] \longleftrightarrow False$
⟨*proof*⟩

lemma *par-not-refl* [*simp*]: $p \perp p \longleftrightarrow False$
⟨*proof*⟩

lemma *par-Cons-iff*:
$(i \# ps) \perp (j \# qs) \longleftrightarrow (i \neq j \vee ps \perp qs)$
⟨*proof*⟩

Simplification rules on *poss*.

lemma *Nil-in-poss* [*simp*]: $[] \in poss\ t$
⟨*proof*⟩

lemma *poss-Cons* [*simp*]:
$i \# p \in poss\ t \Longrightarrow [i] \in poss\ t$
⟨*proof*⟩

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

lemma *poss-append-poss*:
$p\ @\ q \in poss\ t \longleftrightarrow p \in poss\ t \wedge q \in poss\ (t\ |\text{-}\ p)$
⟨*proof*⟩

Simplification rules on *hole-pos*

lemma *hole-pos-map-vars* [*simp*]:
hole-pos (*map-vars-ctxt f C*) = *hole-pos C*
⟨*proof*⟩

lemma *hole-pos-in-ctxt-apply* [*simp*]: *hole-pos* $C \in poss\ C\langle u\rangle$
⟨*proof*⟩

2.3 Properties of *ground* and *ground-ctxt*

lemma *ground-vars-term-empty* [*simp*]:
ground t $\Longrightarrow$ *vars-term t* = {}
⟨*proof*⟩

lemma *ground-map-term* [*simp*]:
ground (*map-term f h t*) = *ground t*
⟨*proof*⟩

lemma *ground-ctxt-apply* [*simp*]:
ground $C\langle t\rangle \longleftrightarrow$ *ground-ctxt C* $\wedge$ *ground t*
⟨*proof*⟩

lemma *ground-ctxt-comp* [*intro*]:
ground-ctxt C $\Longrightarrow$ *ground-ctxt D* $\Longrightarrow$ *ground-ctxt* ($C \circ_c D$)
⟨*proof*⟩

lemma *ctxt-comp-n-pres-ground* [*intro*]:
ground-ctxt C $\Longrightarrow$ *ground-ctxt* (C^n)
⟨*proof*⟩

lemma *subterm-eq-pres-ground*:
assumes *ground s* **and** $s \trianglerighteq t$
shows *ground t* ⟨*proof*⟩

lemma *ground-substD*:
ground ($l \cdot \sigma$) $\Longrightarrow x \in$ *vars-term l* $\Longrightarrow$ *ground* (σ x)
⟨*proof*⟩

lemma *ground-substI*:
($\bigwedge x.\ x \in$ *vars-term s* $\Longrightarrow$ *ground* (σ x)) $\Longrightarrow$ *ground* ($s \cdot \sigma$)
⟨*proof*⟩

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

lemma *funas-ctxt-apply* [*simp*]:
funas-term $C\langle t\rangle$ = *funas-ctxt C* $\cup$ *funas-term t*
⟨*proof*⟩

lemma *funas-term-map* [*simp*]:
funas-term (*map-term f h t*) = (λ (g, n). (f g, n)) ‘ *funas-term t*
⟨*proof*⟩

lemma *funas-term-subst*:
funas-term ($l \cdot \sigma$) = *funas-term l* $\cup$ ($\bigcup$ (*funas-term* ‘ σ ‘ *vars-term l*))
⟨*proof*⟩

lemma *funas-ctxt-comp* [*simp*]:

$\mathit{funas\text{-}ctxt}\ (C \circ_c D) = \mathit{funas\text{-}ctxt}\ C \cup \mathit{funas\text{-}ctxt}\ D$
⟨*proof*⟩

lemma *ctxt-comp-n-funas* [*simp*]:
$(f, v) \in \mathit{funas\text{-}ctxt}\ (C\hat{\ }n) \Longrightarrow (f, v) \in \mathit{funas\text{-}ctxt}\ C$
⟨*proof*⟩

lemma *ctxt-comp-n-pres-funas* [*intro*]:
$\mathit{funas\text{-}ctxt}\ C \subseteq \mathcal{F} \Longrightarrow \mathit{funas\text{-}ctxt}\ (C\hat{\ }n) \subseteq \mathcal{F}$
⟨*proof*⟩

2.5 Properties on subterm at given position (|-)

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

lemma *ctxt-at-pos-subt-at-pos*:
$p \in \mathit{poss}\ t \Longrightarrow (\mathit{ctxt\text{-}at\text{-}pos}\ t\ p)\langle u\rangle\ |\text{-}\ p = u$
⟨*proof*⟩

lemma *ctxt-at-pos-subt-at-id*:
$p \in \mathit{poss}\ t \Longrightarrow (\mathit{ctxt\text{-}at\text{-}pos}\ t\ p)\langle t\ |\text{-}\ p\rangle = t$
⟨*proof*⟩

lemma *subst-at-ctxt-at-eq-termD*:
assumes $s = t\ p \in \mathit{poss}\ t$
shows $s\ |\text{-}\ p = t\ |\text{-}\ p \land \mathit{ctxt\text{-}at\text{-}pos}\ s\ p = \mathit{ctxt\text{-}at\text{-}pos}\ t\ p$ ⟨*proof*⟩

lemma *subst-at-ctxt-at-eq-termI*:
assumes $p \in \mathit{poss}\ s\ p \in \mathit{poss}\ t$
and $s\ |\text{-}p = t\ |\text{-}\ p$
and $\mathit{ctxt\text{-}at\text{-}pos}\ s\ p = \mathit{ctxt\text{-}at\text{-}pos}\ t\ p$
shows $s = t$ ⟨*proof*⟩

lemma *subt-at-subterm-eq* [*intro*!]:
$p \in \mathit{poss}\ t \Longrightarrow t \trianglerighteq t\ |\text{-}\ p$
⟨*proof*⟩

lemma *subt-at-subterm* [*intro*!]:
$p \in \mathit{poss}\ t \Longrightarrow p \neq [] \Longrightarrow\ t \rhd t\ |\text{-}\ p$
⟨*proof*⟩

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

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

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

$p \notin poss\ s \Longrightarrow s[p \leftarrow t] = s$
⟨*proof*⟩

lemma *replace-term-at-replace-at-conv*:
$p \in poss\ s \Longrightarrow (ctxt\text{-}at\text{-}pos\ s\ p)\langle t\rangle = s[p \leftarrow t]$
⟨*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*]:
$p \leq_p q \Longrightarrow s[q \leftarrow t][p \leftarrow u] = s[p \leftarrow u]$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *adapt-vars-ctxt*[*simp*]: *adapt-vars* (*C* ⟨ *t* ⟩) = (*adapt-vars-ctxt C*) ⟨ *adapt-vars t* ⟩
⟨*proof*⟩

lemma *adapt-vars-subst*[*simp*]: *adapt-vars* ($l \cdot \sigma$) = $l \cdot$ (λ *x*. *adapt-vars* (σ *x*))
⟨*proof*⟩

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

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

2.7.1 Equality on ground terms/contexts by positions and symbols

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

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

lemma *eq-term-by-poss-fun-at*:
assumes *poss s* = *poss t* $\bigwedge$*p*. *p* $\in$ *poss s* $\Longrightarrow$ *fun-at s p* = *fun-at t p*
shows *s* = *t*
⟨*proof*⟩

lemma *eq-ctxt-at-pos-by-poss*:
assumes *p* $\in$ *poss s p* $\in$ *poss t*
and $\bigwedge$ *q*. $\neg$ ($p \leq_p q$) $\Longrightarrow$ *q* $\in$ *poss s* $\longleftrightarrow$ *q* $\in$ *poss t*
and ($\bigwedge$ *q*. *q* $\in$ *poss s* $\Longrightarrow$ $\neg$ ($p \leq_p q$) $\Longrightarrow$ *fun-at s q* = *fun-at t q*)
shows *ctxt-at-pos s p* = *ctxt-at-pos t p* ⟨*proof*⟩

2.8 Misc

lemma *fun-at-hole-pos-ctxt-apply* [*simp*]:
fun-at C⟨*t*⟩ (*hole-pos C*) = *fun-at t* []
⟨*proof*⟩

lemma *vars-term-ctxt-apply* [*simp*]:
$vars\text{-}term\ C\langle t\rangle = vars\text{-}ctxt\ C \cup vars\text{-}term\ t$
⟨*proof*⟩

lemma *map-vars-term-ctxt-apply*:
$map\text{-}vars\text{-}term\ f\ C\langle t\rangle = (map\text{-}vars\text{-}ctxt\ f\ C)\langle map\text{-}vars\text{-}term\ f\ t\rangle$
⟨*proof*⟩

lemma *map-term-replace-at-dist*:
$p \in poss\ s \Longrightarrow (map\text{-}term\ f\ g\ s)[p \leftarrow (map\text{-}term\ f\ g\ t)] = map\text{-}term\ f\ g\ (s[p \leftarrow t])$
⟨*proof*⟩

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

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

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

fun *remove-sum* **where**
$remove\text{-}sum\ (Inl\ q) = q$
$|\ remove\text{-}sum\ (Inr\ q) = q$

List operations

definition *filter-rev-nth* **where**
$filter\text{-}rev\text{-}nth\ P\ xs\ i = length\ (filter\ P\ (take\ (Suc\ i)\ xs)) - 1$

lemma *filter-rev-nth-butlast*:
$\neg\ P\ (last\ xs) \Longrightarrow filter\text{-}rev\text{-}nth\ P\ xs\ i = filter\text{-}rev\text{-}nth\ P\ (butlast\ xs)\ i$
⟨*proof*⟩

lemma *filter-rev-nth-idx*:
assumes $i < length\ xs\ P\ (xs\ !\ i)\ ys = filter\ P\ xs$
shows $xs\ !\ i = ys\ !\ (filter\text{-}rev\text{-}nth\ P\ xs\ i) \wedge filter\text{-}rev\text{-}nth\ P\ xs\ i < length\ ys$
⟨*proof*⟩

primrec *add-elem-list-lists* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list\ list$ **where**
$add\text{-}elem\text{-}list\text{-}lists\ x\ [] = [[x]]$
$|\ add\text{-}elem\text{-}list\text{-}lists\ x\ (y\ \#\ ys) = (x\ \#\ y\ \#\ ys)\ \#\ (map\ ((\#)\ y)\ (add\text{-}elem\text{-}list\text{-}lists$

x ys))

lemma *length-add-elem-list-lists*:
  ys ∈ *set* (*add-elem-list-lists x xs*) ⟹ *length ys* = *Suc* (*length xs*)
  ⟨*proof*⟩

lemma *add-elem-list-listsE*:
  assumes *ys* ∈ *set* (*add-elem-list-lists x xs*)
  shows ∃ *n* ≤ *length xs*. *ys* = *take n xs* @ *x* # *drop n xs* ⟨*proof*⟩

lemma *add-elem-list-listsI*:
  assumes *n* ≤ *length xs ys* = *take n xs* @ *x* # *drop n xs*
  shows *ys* ∈ *set* (*add-elem-list-lists x xs*) ⟨*proof*⟩

lemma *add-elem-list-lists-def'*:
  set (*add-elem-list-lists x xs*) = {*ys* | *ys n*. *n* ≤ *length xs* ∧ *ys* = *take n xs* @ *x* # *drop n xs*}
  ⟨*proof*⟩

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

lemma *list-of-permutation-element-n-conv*:
  assumes *n* ≠ *0*
  shows *set* (*list-of-permutation-element-n x n L*) =
    {*xs* | *xs i*. *i* < *length xs* ∧ (∀ *j* < *length xs*. *j* ≠ *i* ⟶ *xs* ! *j* ∈ *set L*) ∧ *length xs* = *n* ∧ *xs* ! *i* = *x*} (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *list-of-permutation-element-n-iff*:
  set (*list-of-permutation-element-n x n L*) =
    (*if n* = *0 then* {[]} *else* {*xs* | *xs i*. *i* < *length xs* ∧ (∀ *j* < *length xs*. *j* ≠ *i* ⟶ *xs* ! *j* ∈ *set L*) ∧ *length xs* = *n* ∧ *xs* ! *i* = *x*})
⟨*proof*⟩

lemma *list-of-permutation-element-n-conv'*:
  assumes *x* ∈ *set L 0* < *n*
  shows *set* (*list-of-permutation-element-n x n L*) =
      {*xs*. *set xs* ⊆ *insert x* (*set L*) ∧ *length xs* = *n* ∧ *x* ∈ *set xs*}
⟨*proof*⟩

Misc

lemma *in-set-idx*:
  x ∈ *set xs* ⟹ ∃ *i* < *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* < *length xs then xs* ! *i else if i* = *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*: ⋀*x y*. (*x*, *y*) ∈ *r* ⟹ (*f x*, *f y*) ∈ *s*
and *steps*: $(x, y) \in r^+$
shows $(f\ x, f\ y) \in s^+$ ⟨*proof*⟩

lemma *trancl-map-prod-mono*:
map-both f ‘ R^+ ⊆ (*map-both f* ‘ *R*)$^+$
⟨*proof*⟩

lemma *trancl-map-both-Restr*:
assumes *inj-on f X*
shows (*map-both f* ‘ *Restr R X*)$^+$ = *map-both f* ‘ (*Restr R X*)$^+$
⟨*proof*⟩

lemma *inj-on-trancl-map-both*:
assumes *inj-on f* (*fst* ‘ *R* ∪ *snd* ‘ *R*)
shows (*map-both f* ‘ *R*)$^+$ = *map-both f* ‘ R^+
⟨*proof*⟩

lemma *kleene-induct*:
A ⊆ *X* ⟹ *B O X* ⊆ *X* ⟹ *X O C* ⊆ *X* ⟹ B^* *O A O* C^* ⊆ *X*
⟨*proof*⟩

lemma *kleene-trancl-induct*:
A ⊆ *X* ⟹ *B O X* ⊆ *X* ⟹ *X O C* ⊆ *X* ⟹ B^+ *O A O* C^+ ⊆ *X*
⟨*proof*⟩

lemma *rtrancl-Un2-separatorE*:

$B\ O\ A = \{\} \Longrightarrow (A \cup B)^* = A^* \cup A^*\ O\ B^*$
⟨*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*]:
¬ *is-Inl t* ⟷ *is-Inr t*
¬ *is-Inr t* ⟷ *is-Inl t*
⟨*proof*⟩

lemma [*simp*]: *remove-sum* ∘ *Inl* = *id* ⟨*proof*⟩

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

2.9 Ground constructions

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

2.9.1 Ground terms

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

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

lemma *gterm-idx-induct*[*case-names GFun*]:
  assumes $\bigwedge$ *f ts*. ($\bigwedge$ *i*. *i* < *length ts* $\Longrightarrow$ *P* (*ts* ! *i*)) $\Longrightarrow$ *P* (*GFun f ts*)
  shows *P t* ⟨*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 $\Longrightarrow$ *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* $\Longrightarrow$ *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 $\Longrightarrow$ *term-of-gterm* (*gterm-of-term t*) = *t*
  ⟨*proof*⟩

lemma *ground-term-to-gtermD*:

ground t $\Longrightarrow \exists\, t'.\ t =$ *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 $\Longrightarrow$ *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*
⟨*proof*⟩

lemma *vars-term-of-gterm* [*simp*]:
vars-term (*term-of-gterm t*) = {}
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *gposs-map-gterm* [*simp*]:
gposs (*map-gterm f s*) = *gposs s*
⟨*proof*⟩

lemma *poss-gposs-conv*:
poss (*term-of-gterm t*) = *gposs t*
⟨*proof*⟩

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

lemma *gposs-to-poss*:
  p ∈ *gposs t* ⟹ *p* ∈ *poss* (*term-of-gterm t*)
  ⟨*proof*⟩

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

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

lemma *gfun-at-gterm-of-term* [*simp*]:
  ground s ⟹ *map-option exInl* (*fun-at s p*) = *gfun-at* (*gterm-of-term s*) *p*
⟨*proof*⟩

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

lemma *gfun-at-None-ngposs-iff*: *gfun-at s p* = *None* ⟷ *p* ∉ *gposs s*
  ⟨*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*) | *p*. *p* ∈ *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* ⟷ *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* ⇒ *pos* ⇒ *'f gterm* **where**
  gsubt-at s [] = *s* |
  gsubt-at (*GFun f ss*) (*i* # *p*) = *gsubt-at* (*ss* ! *i*) *p*

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

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

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

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

lemma *gposs-gsubst-at-subst-at-eq* [*simp*]:
assumes $p \in gposs\ s$
shows $gposs\ (gsubt\text{-}at\ s\ p) = poss\ (term\text{-}of\text{-}gterm\ s\ |\text{-}\ p)$ ⟨*proof*⟩

lemma *gpos-append-gposs*:
assumes $p \in gposs\ t$ **and** $q \in gposs\ (gsubt\text{-}at\ t\ p)$
shows $p\ @\ q \in gposs\ t$
⟨*proof*⟩

Replace terms at position

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

lemma *replace-gterm-at-not-poss* [*simp*]:
$p \notin gposs\ s \Longrightarrow s[p \leftarrow t]_G = s$
⟨*proof*⟩

lemma *parallel-replace-gterm-commute* [*ac-simps*]:
$p \perp q \Longrightarrow s[p \leftarrow t]_G[q \leftarrow u]_G = s[q \leftarrow u]_G[p \leftarrow t]_G$
⟨*proof*⟩

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

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

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

$p \neq [] \Longrightarrow$ *groot-sym* $s[p \leftarrow t]_G =$ *groot-sym s*
⟨*proof*⟩

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

lemma *replace-gterm-conv*:
$p \in$ *gposs s* $\Longrightarrow$ (*term-of-gterm s*)$[p \leftarrow$ (*term-of-gterm t*)$]$ = *term-of-gterm* ($s[p$ $\leftarrow t]_G$)
⟨*proof*⟩

2.9.2 Tree domains

type-synonym *gdomain* = *unit gterm*

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

lemma *gdomain-id*:
gdomain t = *t*
⟨*proof*⟩

lemma *gdomain-gsubt* [*simp*]:
assumes $p \in$ *gposs t*
shows *gdomain* (*gsubt-at t p*) = *gsubt-at* (*gdomain t*) *p*
⟨*proof*⟩

Union of tree domains

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

lemma *gposs-gunion* [*simp*]:
gposs (*gunion s t*) = *gposs s* ∪ *gposs t*
⟨*proof*⟩

lemma *gunion-unit* [*simp*]:
gunion s (*GFun* () []) = *s gunion* (*GFun* () []) *s* = *s*
⟨*proof*⟩

lemma *gunion-gsubt-at-nt-poss1*:
assumes $p \in$ *gposs s* **and** $p \notin$ *gposs t*
shows *gsubt-at* (*gunion s t*) *p* = *gsubt-at s p*
⟨*proof*⟩

lemma *gunion-gsubt-at-nt-poss2*:
assumes $p \in$ *gposs t* **and** $p \notin$ *gposs s*
shows *gsubt-at* (*gunion s t*) *p* = *gsubt-at t p*

⟨proof⟩

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

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

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

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

lemma *gunion-idemp* [*simp*]:
$gunion\ s\ s = s$
⟨proof⟩

definition *gunions* :: *gdomain list* ⇒ *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*:
$map\text{-}gterm\ f\ (glabel\ h\ t) = glabel\ (f \circ h)\ t$
⟨proof⟩

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

gfun-at (*glabel f t*) *p* = (*if p* ∈ *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* ∘ *gfun-at t*) (*gdomain t*) = *map-gterm* (*f* ∘ *Some*) *t*
⟨*proof*⟩

lemma *gfun-at-nongposs* [*simp*]:
p ∉ *gposs t* ⟹ *gfun-at t p* = *None*
⟨*proof*⟩

lemma *gfun-at-poss*:
p ∈ *gposs t* ⟹ ∃ *f*. *gfun-at t p* = *Some f*
⟨*proof*⟩

lemma *gfun-at-possE*:
assumes *p* ∈ *gposs t*
obtains *f* **where** *gfun-at t p* = *Some f*
⟨*proof*⟩

lemma *gfun-at-poss-gpossD*:
gfun-at t p = *Some f* ⟹ *p* ∈ *gposs t*
⟨*proof*⟩

function symbols of a ground term

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

lemma *funas-gterm-gterm-of-term*:
ground t ⟹ *funas-gterm* (*gterm-of-term t*) = *funas-term t*
⟨*proof*⟩

lemma *funas-term-of-gterm-conv*:
funas-term (*term-of-gterm t*) = *funas-gterm t*
⟨*proof*⟩

lemma *funas-gterm-map-gterm*:
assumes *funas-gterm t* ⊆ $\mathcal{F}$
shows *funas-gterm* (*map-gterm f t*) ⊆ (λ (*h*, *n*). (*f h*, *n*)) ‘ $\mathcal{F}$
⟨*proof*⟩

lemma *gterm-of-term-inj*:
assumes ⋀ *t*. *t* ∈ *S* ⟹ *ground t*
shows *inj-on gterm-of-term S*
⟨*proof*⟩

lemma *funas-gterm-gsubt-at-subseteq*:
assumes $p \in gposs\ s$
shows $funas\text{-}gterm\ (gsubt\text{-}at\ s\ p) \subseteq funas\text{-}gterm\ s$ ⟨*proof*⟩

lemma *finite-funas-gterm*: $finite\ (funas\text{-}gterm\ t)$
⟨*proof*⟩

ground term set

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

lemma *gterms-mono*:
$\mathcal{G} \subseteq \mathcal{F} \Longrightarrow gterms\ \mathcal{G} \subseteq gterms\ \mathcal{F}$
⟨*proof*⟩

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

lemma $\mathcal{T}_G$*-sound*:
$s \in \mathcal{T}_G\ \mathcal{F} \Longrightarrow funas\text{-}gterm\ s \subseteq \mathcal{F}$
⟨*proof*⟩

lemma $\mathcal{T}_G$*-complete*:
$funas\text{-}gterm\ s \subseteq \mathcal{F} \Longrightarrow s \in \mathcal{T}_G\ \mathcal{F}$
⟨*proof*⟩

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

lemma $\mathcal{T}_G$*-equivalent-def*:
$\mathcal{T}_G\ \mathcal{F} = gterms\ \mathcal{F}$
⟨*proof*⟩

lemma $\mathcal{T}_G$*-intersection* [*simp*]:
$s \in \mathcal{T}_G\ \mathcal{F} \Longrightarrow s \in \mathcal{T}_G\ \mathcal{G} \Longrightarrow s \in \mathcal{T}_G\ (\mathcal{F} \cap \mathcal{G})$
⟨*proof*⟩

lemma $\mathcal{T}_G$*-mono*:
$\mathcal{G} \subseteq \mathcal{F} \Longrightarrow \mathcal{T}_G\ \mathcal{G} \subseteq \mathcal{T}_G\ \mathcal{F}$
⟨*proof*⟩

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

definition *funas-grel* **where**

funas-grel $\mathcal{R}$ = $\bigcup$ ((λ (*s*, *t*). *funas-gterm s* ∪ *funas-gterm t*) ' $\mathcal{R}$)

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

context
includes *fset.lifting*
begin

lift-definition *fCollect* :: (′*a* ⇒ *bool*) ⇒ ′*a fset* **is** λ *P*. *if finite* (*Collect P*) *then Collect P else* {}
  ⟨*proof*⟩

lift-definition *fSigma* :: ′*a fset* ⇒ (′*a* ⇒ ′*b fset*) ⇒ (′*a* × ′*b*) *fset* **is** *Sigma*
  ⟨*proof*⟩

lift-definition *is-fempty* :: ′*a fset* ⇒ *bool* **is** *Set.is-empty* ⟨*proof*⟩
lift-definition *fremove* :: ′*a* ⇒ ′*a fset* ⇒ ′*a fset* **is** *Set.remove*
  ⟨*proof*⟩

lift-definition *finj-on* :: (′*a* ⇒ ′*b*) ⇒ ′*a fset* ⇒ *bool* **is** *inj-on* ⟨*proof*⟩
lift-definition *the-finv-into* :: ′*a fset* ⇒ (′*a* ⇒ ′*b*) ⇒ ′*b* ⇒ ′*a* **is** *the-inv-into* ⟨*proof*⟩

lemma *fCollect-memberI* [*intro*!]:
  finite (*Collect P*) ⟹ *P x* ⟹ *x* |∈| *fCollect P*
  ⟨*proof*⟩

lemma *fCollect-member* [*iff*]:
  x |∈| *fCollect P* ⟷ *finite* (*Collect P*) ∧ *P x*
  ⟨*proof*⟩

lemma *fCollect-cong*: (⋀*x*. *P x* = *Q x*) ⟹ *fCollect P* = *fCollect Q*
  ⟨*proof*⟩
end

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

syntax (*ASCII*)
  -fCollect :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ ′*a set* ((*1*{(-/|:| -)./ -}))
syntax
  -fCollect :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ ′*a set* ((*1*{(-/ |∈| -)./ -}))
translations

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

syntax (*ASCII*)
-*fBall* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3ALL* (-/|:|-)./ -) [*0*, *0*, *10*] *10*)
-*fBex* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3EX* (-/|:|-)./ -) [*0*, *0*, *10*] *10*)

syntax (*input*)
-*fBall* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*! (-/|:|-)./ -) [*0*, *0*, *10*] *10*)
-*fBex* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3?* (-/|:|-)./ -) [*0*, *0*, *10*] *10*)

syntax
-*fBall* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∀ (-/|∈|-)./ -) [*0*, *0*, *10*] *10*)
-*fBex* :: *pttrn* ⇒ ′*a set* ⇒ *bool* ⇒ *bool* ((*3*∃ (-/|∈|-)./ -) [*0*, *0*, *10*] *10*)

translations
∀ x|∈|A. P ⇌ *CONST fBall A* (λx. P)
∃ x|∈|A. P ⇌ *CONST fBex A* (λx. P)

syntax (*ASCII* **output**)
-*setlessfAll* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3ALL* -|<|-./ -) [*0*, *0*, *10*] *10*)
-*setlessfEx* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3EX* -|<|-./ -) [*0*, *0*, *10*] *10*)
-*setlefAll* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3ALL* -|<=|-./ -) [*0*, *0*, *10*] *10*)
-*setlefEx* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3EX* -|<=|-./ -) [*0*, *0*, *10*] *10*)

syntax
-*setlessfAll* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3*∀ -|⊂|-./ -) [*0*, *0*, *10*] *10*)
-*setlessfEx* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3*∃ -|⊂|-./ -) [*0*, *0*, *10*] *10*)
-*setlefAll* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3*∀ -|⊆|-./ -) [*0*, *0*, *10*] *10*)
-*setlefEx* :: [*idt*, ′*a*, *bool*] ⇒ *bool* ((*3*∃ -|⊆|-./ -) [*0*, *0*, *10*] *10*)

translations
∀ A|⊂|B. P ⇀ ∀ A. A |⊂| B ⟶ P
∃ A|⊂|B. P ⇀ ∃ A. A |⊂| B ∧ P
∀ A|⊆|B. P ⇀ ∀ A. A |⊆| B ⟶ P
∃ A|⊆|B. P ⇀ ∃ A. A |⊆| B ∧ P

syntax
-*fSetcompr* :: ′*a* ⇒ *idts* ⇒ *bool* ⇒ ′*a fset* ((*1*{|- |/-./ -|}))

⟨*ML*⟩

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

notation
ffUnion (|⋃|)

context
includes *fset.lifting*
begin

lemma *right-total-cr-fset* [*transfer-rule*]:
right-total cr-fset
⟨*proof*⟩

lemma *bi-unique-cr-fset* [*transfer-rule*]:
bi-unique cr-fset
⟨*proof*⟩

lemma *right-total-pcr-fset-eq* [*transfer-rule*]:
right-total (*pcr-fset* (=))
⟨*proof*⟩

lemma *bi-unique-pcr-fset* [*transfer-rule*]:
bi-unique (*pcr-fset* (=))
⟨*proof*⟩

lemma *set-fset-of-list-transfer* [*transfer-rule*]:
rel-fun (*list-all2 A*) (*pcr-fset A*) *set fset-of-list*
⟨*proof*⟩

lemma *fCollectD*: a |∈| {|x . P x|} $\Longrightarrow$ P a
⟨*proof*⟩

lemma *fCollectI*: P a $\Longrightarrow$ *finite* (*Collect P*) $\Longrightarrow$ a |∈| {| x. P x|}
⟨*proof*⟩

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

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

lemma *fset-image-conv*:
{f x | x. x |∈| T} = *fset* (f |'| T)
⟨*proof*⟩

lemma *fimage-def*:
f |'| A = {| y. $\exists x$|∈|A. $y = f\ x$ |}
⟨*proof*⟩

lemma *ffilter-simp*: *ffilter P A* = {*a* |∈| *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* ⇒ *'b fset* ⇒ (*'a* × *'b*) *fset* (**infixr** |×| *80*)
where *A* |×| *B* ≡ *fSigma A* (λ-. *B*)

lemma *fSigma-repeq*:
fset (*A* |×| *B*) = *fset A* × *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 *fimage-paired-Times* = *image-paired-Times*[*Transfer.transferred*]
lemmas *fproduct-swap* = *product-swap*[*Transfer.transferred*]
lemmas *fswap-product* = *swap-product*[*Transfer.transferred*]
lemmas *fsubset-fst-snd* = *subset-fst-snd*[*Transfer.transferred*]
lemmas *map-prod-ftimes* = *map-prod-times*[*Transfer.transferred*]

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

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

— Dealing with transitive closure

lift-definition *ftrancl* :: $('a \times 'a)\ fset \Rightarrow ('a \times 'a)\ fset$ ((-|⁺|) [*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.transfer*
lemmas *converse-ftranclE* = *converse-tranclE*[*Transfer.transferred*]
lemma *in-ftrancl-UnI*:
  $x\ |{\in}|\ R|^{+}| \vee x\ |{\in}|\ S|^{+}| \Longrightarrow x\ |{\in}|\ (R\ |{\cup}|\ S)|^{+}|$
  ⟨*proof*⟩

lemma *ftranclD*:
  $(x,\ y)\ |{\in}|\ R|^{+}| \Longrightarrow \exists\, z.\ (x,\ z)\ |{\in}|\ R \wedge (z = y \vee (z,\ y)\ |{\in}|\ R|^{+}|)$
  ⟨*proof*⟩

lemma *ftranclD2*:
  $(x,\ y)\ |{\in}|\ R|^{+}| \Longrightarrow \exists\, z.\ (x = z \vee (x,\ z)\ |{\in}|\ R|^{+}|) \wedge (z,\ y)\ |{\in}|\ R$
  ⟨*proof*⟩

lemma *not-ftrancl-into*:
$(x, z) \mid\notin\mid r|^{+}| \Longrightarrow (y, z) \mid\in\mid r \Longrightarrow (x, y) \mid\notin\mid r|^{+}|$
⟨*proof*⟩
lemmas *ftrancl-map-both-fRestr* = *trancl-map-both-Restr*[*Transfer.transferred*]
lemma *ftrancl-map-both-fsubset*:
finj-on f X $\Longrightarrow$ *R* |⊆| *X* |×| *X* $\Longrightarrow$ (*map-both f* |'| *R*)$|^{+}|$ = *map-both f* |'| *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*:
fset-of-list [*i*..<*j*] = *fCollect* (λ *n*. $i \le n \wedge n < j$)
⟨*proof*⟩

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

lift-definition *fId-on* :: '*a fset* ⇒ ('*a* × '*a*) *fset* **is** *Id-on*
⟨*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-onE* [*elim*!] = *Id-onE* [*Transfer.transferred*]
lemmas *fId-on-iff* = *Id-on-iff* [*Transfer.transferred*]
lemmas *fId-on-fsubset-fTimes* = *Id-on-subset-Times* [*Transfer.transferred*]

lift-definition *fconverse* :: ('*a* × '*b*) *fset* ⇒ ('*b* × '*a*) *fset* ((-$|^{-1}|$) [*1000*] *999*) **is** *converse* ⟨*proof*⟩

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

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

lemma *finj-Inl-Inr*:
finj-on Inl A finj-on Inr A
⟨*proof*⟩
lemma *finj-CInl-CInr*:
finj-on CInl A finj-on CInr A
⟨*proof*⟩

lemma *finj-Some*:
finj-on Some A
⟨*proof*⟩

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

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

lemmas *fImage-mono* = *Image-mono*[*Transfer.transferred*]
lemmas *fImage-fUN* = *Image-UN*[*Transfer.transferred*]
lemmas *fUN-fImage* = *UN-Image*[*Transfer.transferred*]
lemmas *fSigma-fImage* = *Sigma-Image*[*Transfer.transferred*]

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

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

lemma *rel-set-cr-fset*:
rel-set cr-fset = (λ *A B*. *A* = *fset* ‘ *B*)
⟨*proof*⟩
lemma *pcr-fset-cr-fset*:
pcr-fset cr-fset = (λ *x y*. *x* = *fset* (*fset* |‘| *y*))
⟨*proof*⟩

lemma *sorted-list-of-fset-id*:
sorted-list-of-fset x = *sorted-list-of-fset y* ⟹ *x* = *y*
⟨*proof*⟩

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

lemma *finite-ntrancl*:
finite R ⟹ *finite* (*ntrancl n R*)
⟨*proof*⟩

lift-definition *nftrancl* :: *nat* ⇒ (′*a* × ′*a*) *fset* ⇒ (′*a* × ′*a*) *fset* **is** *ntrancl*
⟨*proof*⟩

lift-definition *frelcomp* :: (*'a* × *'b*) *fset* ⇒ (*'b* × *'c*) *fset* ⇒ (*'a* × *'c*) *fset* (**infixr** |*O*| *75*) **is** *relcomp*
⟨*proof*⟩

lemmas *frelcompE*[*elim*!] = *relcompE*[*Transfer.transferred*]
lemmas *frelcompI*[*intro*] = *relcompI*[*Transfer.transferred*]
lemma *fId-on-frelcomp-id*:
fst |'| *R* |⊆| *S* ⟹ *fId-on S* |*O*| *R* = *R*
⟨*proof*⟩
lemma *fId-on-frelcomp-id2*:
snd |'| *R* |⊆| *S* ⟹ *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* ⇒ *'f fset* **is** *funs-term* ⟨*proof*⟩
lift-definition *fvars-term* :: (*'f*, *'v*) *term* ⇒ *'v fset* **is** *vars-term* ⟨*proof*⟩
lift-definition *fvars-ctxt* :: (*'f*, *'v*) *ctxt* ⇒ *'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* [*simp*] = *ground-vars-term-empty*[*Transfer.transferred*]

lemma *ffuns-term-Var* [*simp*]: *ffuns-term* (*Var x*) = {||}
⟨*proof*⟩
lemma *fffuns-term-Fun* [*simp*]: *ffuns-term* (*Fun f ts*) = |⋃| (*ffuns-term* |'| *fset-of-list ts*) |∪| {|*f*|}
⟨*proof*⟩

lemma *fvars-term-Var* [*simp*]: *fvars-term* (*Var x*) = {|*x*|}
⟨*proof*⟩
lemma *fvars-term-Fun* [*simp*]: *fvars-term* (*Fun f ts*) = |⋃| (*fvars-term* |'| *fset-of-list 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 ≤ *P* ⟹ *finite* (*Collect P*) ⟹ *finite* (*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'* = *sorted-list-of-fset* (*ex S*) *in*
let T' = *sorted-list-of-fset* (*ex T*) *in*
S' < *T'*)

instance ⟨*proof*⟩
end

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

2.9.3 Ground context

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

fun *gctxt-apply-term* :: *'f gctxt* ⇒ *'f gterm* ⇒ *'f gterm* (-⟨-⟩$_G$ [*1000*, *0*] *1000*) **where**
  $\Box_G\langle s\rangle_G = s$ |
  (*GMore f ss1 C ss2*)$\langle s\rangle_G$ = *GFun f* (*ss1* @ $C\langle s\rangle_G$ # *ss2*)

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

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

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

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

interpretation *ctxt-monoid-mult*: *monoid-mult* $\Box_G$ ($\circ_{Gc}$)
⟨*proof*⟩

instantiation *gctxt* :: (*type*) *monoid-mult*
begin
  definition [*simp*]: *1* = $\Box_G$
  definition [*simp*]: (∗) = ($\circ_{Gc}$)
  instance ⟨*proof*⟩
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 $\Box_G = \Box$

| *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 $\Box = \Box_G$
| *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* ⟹ *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 ⟹ *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 ⟹ *gctxt-of-ctxt* (*map-ctxt f g C*) = *map-gctxt f* (*gctxt-of-ctxt C*)
⟨*proof*⟩

lemma *map-gctxt-nempty* [*simp*]:
$C \neq \Box_G \Longrightarrow$ *map-gctxt f C* $\neq \Box_G$
⟨*proof*⟩

lemma *gctxt-set-funs-ctxt*:
gfuns-ctxt C = *funs-ctxt* (*ctxt-of-gctxt C*)
⟨*proof*⟩

lemma *ctxt-set-funs-gctxt*:
assumes *ground-ctxt C*
shows *gfuns-ctxt* (*gctxt-of-ctxt C*) = *funs-ctxt C*
⟨*proof*⟩

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

$vars\text{-}ctxt\ (ctxt\text{-}of\text{-}gctxt\ C) = \{\}$
⟨*proof*⟩

lemma *vars-ctxt-of-gctxt-subseteq* [*simp*]:
$vars\text{-}ctxt\ (ctxt\text{-}of\text{-}gctxt\ C) \subseteq Q \longleftrightarrow True$
⟨*proof*⟩

lemma *term-of-gterm-ctxt-apply-ground* [*simp*]:
$term\text{-}of\text{-}gterm\ s = C\langle l\rangle \Longrightarrow ground\text{-}ctxt\ C$
$term\text{-}of\text{-}gterm\ s = C\langle l\rangle \Longrightarrow ground\ l$
⟨*proof*⟩

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

lemma *gctxt-compose-HoleE*:
$C \circ_{Gc} D = \Box_G \Longrightarrow C = \Box_G$
$C \circ_{Gc} D = \Box_G \Longrightarrow D = \Box_G$
⟨*proof*⟩

lemma *nempty-ground-ctxt-gctxt* [*simp*]:
$C \neq \Box \Longrightarrow ground\text{-}ctxt\ C \Longrightarrow gctxt\text{-}of\text{-}ctxt\ C \neq \Box_G$
⟨*proof*⟩

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

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

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

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

lemma *term-of-gterm-ctxt-apply* [*simp*]:
$term\text{-}of\text{-}gterm\ s = C\langle l\rangle \Longrightarrow (gctxt\text{-}of\text{-}ctxt\ C)\langle gterm\text{-}of\text{-}term\ l\rangle_G = s$
⟨*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*

$\langle proof \rangle$

lemma *ctxt-of-pos-gterm* [*simp*]:
$p \in gposs\ t \Longrightarrow ctxt\text{-}at\text{-}pos\ (term\text{-}of\text{-}gterm\ t)\ p = ctxt\text{-}of\text{-}gctxt\ (gctxt\text{-}at\text{-}pos\ t\ p)$
$\langle proof \rangle$

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

The position of the hole in a context is uniquely determined

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

lemma *ghole-pos-gctxt-at-pos* [*simp*]:
$p \in gposs\ t \Longrightarrow ghole\text{-}pos\ (gctxt\text{-}at\text{-}pos\ t\ p) = p$
$\langle proof \rangle$

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

lemma *ghole-pos-in-apply*:
$ghole\text{-}pos\ C = p \Longrightarrow p \in gposs\ C\langle u\rangle_G$
$\langle proof \rangle$

lemma *ground-hole-pos-to-ghole*:
$ground\text{-}ctxt\ C \Longrightarrow ghole\text{-}pos\ (gctxt\text{-}of\text{-}ctxt\ C) = hole\text{-}pos\ C$
$\langle proof \rangle$

lemma *gsubst-at-gctxt-at-eq-gtermD*:
assumes $s = t$ $p \in gposs\ t$
shows $gsubt\text{-}at\ s\ p = gsubt\text{-}at\ t\ p \wedge gctxt\text{-}at\text{-}pos\ s\ p = gctxt\text{-}at\text{-}pos\ t\ p$ $\langle proof \rangle$

lemma *gsubst-at-gctxt-at-eq-gtermI*:
assumes $p \in gposs\ s$ $p \in gposs\ t$
and $gsubt\text{-}at\ s\ p = gsubt\text{-}at\ t\ p$
and $gctxt\text{-}at\text{-}pos\ s\ p = gctxt\text{-}at\text{-}pos\ t\ p$
shows $s = t$ $\langle proof \rangle$

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

lemma *gfun-at-gctxt-at-pos-not-after*:
assumes $p \in gposs\ t\ q \in gposs\ t\ \neg\ (p \leq_p q)$
shows $gfun\text{-}at\ (gctxt\text{-}at\text{-}pos\ t\ p)\langle v\rangle_G\ q = gfun\text{-}at\ t\ q$ ⟨*proof*⟩

lemma *gpos-less-eq-append* [*simp*]: $p \leq_p (p\ @\ q)$
⟨*proof*⟩

lemma *gposs-ConsE* [*elim*]:
assumes $i\ \#\ p \in gposs\ t$
obtains *f ts* **where** $t = GFun\ f\ ts\ ts \neq []\ i < length\ ts\ p \in gposs\ (ts\ !\ i)$ ⟨*proof*⟩

lemma *gposs-gctxt-at-pos-not-after*:
assumes $p \in gposs\ t\ q \in gposs\ t\ \neg\ (p \leq_p q)$
shows $q \in gposs\ (gctxt\text{-}at\text{-}pos\ t\ p)\langle v\rangle_G \longleftrightarrow q \in gposs\ t$ ⟨*proof*⟩

lemma *gposs-gctxt-at-pos*:
$p \in gposs\ t \Longrightarrow gposs\ (gctxt\text{-}at\text{-}pos\ t\ p)\langle v\rangle_G = \{q.\ q \in gposs\ t \wedge \neg\ (p \leq_p q)\} \cup$ $(@)\ p\ `\ gposs\ v$
⟨*proof*⟩

lemma *eq-gctxt-at-pos*:
assumes $p \in gposs\ s\ p \in gposs\ t$
and $\bigwedge q.\ \neg\ (p \leq_p q) \Longrightarrow q \in gposs\ s \longleftrightarrow q \in gposs\ t$
and $(\bigwedge q.\ q \in gposs\ s \Longrightarrow \neg\ (p \leq_p q) \Longrightarrow gfun\text{-}at\ s\ q = gfun\text{-}at\ t\ q)$
shows $gctxt\text{-}at\text{-}pos\ s\ p = gctxt\text{-}at\text{-}pos\ t\ p$ ⟨*proof*⟩

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*]:
$ground\text{-}ctxt\ C \Longrightarrow funas\text{-}gctxt\ (gctxt\text{-}of\text{-}ctxt\ C) = funas\text{-}ctxt\ C$
⟨*proof*⟩

lemma *funas-ctxt-of-gctxt-conv* [*simp*]:
$funas\text{-}ctxt\ (ctxt\text{-}of\text{-}gctxt\ C) = funas\text{-}gctxt\ C$
⟨*proof*⟩

lemma *inj-gctxt-of-ctxt-on-ground*:
$inj\text{-}on\ gctxt\text{-}of\text{-}ctxt\ (Collect\ ground\text{-}ctxt)$
⟨*proof*⟩

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

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

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

2.9.4 Multihole context closure

Computing the multihole context closure of a given relation

inductive-set *gmctxt-cl* :: (′*f* × *nat*) *set* ⇒ ′*f gterm rel* ⇒ ′*f gterm rel* **for** $\mathcal{F}$ $\mathcal{R}$
where
base [*intro*]: (*s*, *t*) ∈ $\mathcal{R}$ ⟹ (*s*, *t*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
| *step* [*intro*]: *length ss* = *length ts* ⟹ (∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$) ⟹ (*f*, *length ss*) ∈ $\mathcal{F}$ ⟹
(*GFun f ss*, *GFun f ts*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$

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

lemma *gmctxt-cl-refl*:
funas-gterm t ⊆ $\mathcal{F}$ ⟹ (*t*, *t*) ∈ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
⟨*proof*⟩

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

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

lemma *gmctxt-cl-mono-rel*:
assumes $\mathcal{P}$ ⊆ $\mathcal{R}$ **shows** *gmctxt-cl* $\mathcal{F}$ $\mathcal{P}$ ⊆ *gmctxt-cl* $\mathcal{F}$ $\mathcal{R}$
⟨*proof*⟩

definition *gcomp-rel* :: (′*f* × *nat*) *set* ⇒ ′*f gterm rel* ⇒ ′*f gterm rel* ⇒ ′*f gterm rel* **where**
gcomp-rel $\mathcal{F}$ *R S* = (*R O gmctxt-cl* $\mathcal{F}$ *S*) ∪ (*gmctxt-cl* $\mathcal{F}$ *R O S*)

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

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

⟨*proof*⟩

2.9.5 Signature closed property

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

lemma *all-ctxt-closedI*:
assumes ⋀ *f ss ts*. (*f*, *length ss*) ∈ $\mathcal{F}$ ⟹ *length ss* = *length ts* ⟹
(∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *r*) ⟹ (*GFun f ss*, *GFun f ts*) ∈ *r*
shows *all-ctxt-closed* $\mathcal{F}$ *r* ⟨*proof*⟩

lemma *all-ctxt-closedD*:
all-ctxt-closed F r ⟹ (*f*, *length ss*) ∈ *F* ⟹ *length ss* = *length ts* ⟹
(∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ *r*) ⟹ (*GFun f ss*, *GFun f ts*) ∈ *r*
⟨*proof*⟩

lemma *all-ctxt-closed-refl-on*:
assumes *all-ctxt-closed* $\mathcal{F}$ *r s* ∈ $\mathcal{T}_G$ $\mathcal{F}$
shows (*s*, *s*) ∈ *r* ⟨*proof*⟩

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

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

2.9.6 Transitive closure preserves *all-ctxt-closed*

induction scheme for transitive closures of lists

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

lemma *trancl-list-appendI* [*simp*, *intro*]:
(*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$ ⟹ (*x*, *y*) ∈ $\mathcal{R}$ ⟹ (*x* # *xs*, *y* # *ys*) ∈ *trancl-list* $\mathcal{R}$
⟨*proof*⟩

lemma *trancl-list-append-tranclI* [*intro*]:
(*x*, *y*) ∈ $\mathcal{R}^+$ ⟹ (*xs*, *ys*) ∈ *trancl-list* $\mathcal{R}$ ⟹ (*x* # *xs*, *y* # *ys*) ∈ *trancl-list* $\mathcal{R}$
⟨*proof*⟩

lemma *trancl-list-conv*:
$(xs,\ ys) \in trancl\text{-}list\ \mathcal{R} \longleftrightarrow length\ xs = length\ ys \wedge (\forall\ i < length\ ys.\ (xs\ !\ i,\ ys\ !\ i) \in \mathcal{R}^+)$ (**is** *?Ls* $\longleftrightarrow$ *?Rs*)
$\langle proof \rangle$

lemma *trancl-list-induct* [*consumes 2*, *case-names base step*]:
assumes $length\ ss = length\ ts\ \forall\ i < length\ ts.\ (ss\ !\ i,\ ts\ !\ i) \in \mathcal{R}^+$
and $\bigwedge xs\ ys.\ length\ xs = length\ ys \Longrightarrow \forall\ i < length\ ys.\ (xs\ !\ i,\ ys\ !\ i) \in \mathcal{R} \Longrightarrow P\ xs\ ys$
and $\bigwedge xs\ ys\ i\ z.\ length\ xs = length\ ys \Longrightarrow \forall\ i < length\ ys.\ (xs\ !\ i,\ ys\ !\ i) \in \mathcal{R}^+ \Longrightarrow P\ xs\ ys$
$\Longrightarrow i < length\ ys \Longrightarrow (ys\ !\ i,\ z) \in \mathcal{R} \Longrightarrow P\ xs\ (ys[i := z])$
shows $P\ ss\ ts$ $\langle proof \rangle$

lemma *trancl-list-all-step-induct* [*consumes 2*, *case-names base step*]:
assumes $length\ ss = length\ ts\ \forall\ i < length\ ts.\ (ss\ !\ i,\ ts\ !\ i) \in \mathcal{R}^+$
and *base*: $\bigwedge xs\ ys.\ length\ xs = length\ ys \Longrightarrow \forall\ i < length\ ys.\ (xs\ !\ i,\ ys\ !\ i) \in \mathcal{R} \Longrightarrow P\ xs\ ys$
and *steps*: $\bigwedge xs\ ys\ zs.\ length\ xs = length\ ys \Longrightarrow length\ ys = length\ zs \Longrightarrow$
$\forall\ i < length\ zs.\ (xs\ !\ i,\ ys\ !\ i) \in \mathcal{R}^+ \Longrightarrow \forall\ i < length\ zs.\ (ys\ !\ i,\ zs\ !\ i) \in \mathcal{R} \vee ys\ !\ i = zs\ !\ i \Longrightarrow$
$P\ xs\ ys \Longrightarrow P\ xs\ zs$
shows $P\ ss\ ts$ $\langle proof \rangle$

lemma *all-ctxt-closed-trancl*:
assumes $all\text{-}ctxt\text{-}closed\ \mathcal{F}\ \mathcal{R}\ \mathcal{R} \subseteq \mathcal{T}_G\ \mathcal{F} \times \mathcal{T}_G\ \mathcal{F}$
shows $all\text{-}ctxt\text{-}closed\ \mathcal{F}\ (\mathcal{R}^+)$
$\langle proof \rangle$

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

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

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

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

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

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

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

lemma *step-mono*:
  $(G,\ B) \vdash (G',\ B') \Longrightarrow (H \cup G,\ B) \vdash (H \cup G',\ B')$
  ⟨*proof*⟩

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

lemma *inv-start*:
  shows *invariant* (*infer0*, {})
  ⟨*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 $B = saturate$
⟨*proof*⟩

lemma *step-sound*:
  $(infer0,\ \{\}) \vdash (\{\},\ B) \Longrightarrow B = saturate$
  ⟨*proof*⟩

end

lemma *horn-infer0-union*:
  $horn.infer0\ (\mathcal{H}_1 \cup \mathcal{H}_2) = horn.infer0\ \mathcal{H}_1 \cup horn.infer0\ \mathcal{H}_2$
  ⟨*proof*⟩

lemma *horn-infer1-union*:
  $horn.infer1\ (\mathcal{H}_1 \cup \mathcal{H}_2)\ x\ B = horn.infer1\ \mathcal{H}_1\ x\ B \cup horn.infer1\ \mathcal{H}_2\ x\ B$
  ⟨*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 (λ*xa*. *case-option None* (*f xa*)) *xs None* = *None*
⟨*proof*⟩

lemma *saturate-fold-mono* [*partial-function-mono*]:
option.mono-body (λ*f*. *fold* (λ*x*. *case-option None* (λ*y*. *f* (*x*, *y*))) *xs b*)
⟨*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 (λ*x*. *case-option None* (*saturate-rec x*)) (*infer1-impl x bs*) (*Some* (*x* # *bs*)))

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

end

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

lemma *saturate-rec-sound*:
saturate-rec x bs = *Some bs'* ⟹ ({*x*}, *set bs*) ⊢ ({}, *set bs'*)
⟨*proof*⟩

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* ⇒ *'a fset* ⇒ *'a list*
begin

lemma *saturate-fold-simp* [*simp*]:
  fold (λ*xa*. *case-option None* (*f xa*)) *xs None* = *None*
  ⟨*proof*⟩

lemma *saturate-fold-mono* [*partial-function-mono*]:
  option.mono-body (λ*f*. *fold* (λ*x*. *case-option None* (λ*y*. *f* (*x*, *y*))) *xs b*)
  ⟨*proof*⟩

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

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

end

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

lemma *saturate-rec-sound*:
  saturate-rec x bs = *Some bs'* ⟹ ({*x*}, *fset bs*) ⊢ ({}, *fset bs'*)
⟨*proof*⟩

lemma *saturate-impl-sound*:
  assumes *saturate-impl* = *Some B'*
  shows *fset B'* = *saturate*
⟨*proof*⟩

lemma *saturate-impl-complete*:
  assumes *finite saturate*
  shows *saturate-impl* ≠ *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*: $('q \times 'q)$ *fset*)

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

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

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

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

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

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

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

3.1.1 Rechability of a term induced by a tree automaton

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

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

ta-der′ $\mathcal{A}$ (*Var p*) = {| *Var q* | *q*. *p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| |}
| *ta-der′* $\mathcal{A}$ (*Fun f ts*) = {| *Var q* | *q*. *q* |∈| *ta-der* $\mathcal{A}$ (*Fun f ts*)|} |∪|
{|*Fun f ss* | *ss*. *length ss* = *length ts* ∧
(∀ *i* < *length ts*. *ss* ! *i* |∈| *ta-der′* $\mathcal{A}$ (*ts* ! *i*))|}

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

abbreviation *ex-rule-state* ≡ *fst* ∘ *groot-sym*
abbreviation *ex-comp-state* ≡ *snd* ∘ *groot-sym*

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

3.1.2 Language acceptance

definition *ta-lang* :: *′q fset* ⇒ (*′q*, *′f*) *ta* ⇒ (*′f*, *′v*) *terms* **where**
[*code del*]: *ta-lang Q* $\mathcal{A}$ = {*adapt-vars t* | *t*. *ground t* ∧ *Q* |∩| *ta-der* $\mathcal{A}$ *t* ≠ {||}}

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

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

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

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

3.1.3 Trimming

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

definition *ta-reachable* :: (*′q*, *′f*) *ta* ⇒ *′q fset* **where**
ta-reachable $\mathcal{A}$ = {|*q*| *q*. ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t* |}

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

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

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

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

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

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

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

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

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

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

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

3.1.4 Mapping over tree automata

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

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

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

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

3.1.5 Product construction (language intersection)

definition *prod-ta-rules* :: ('*q1*,'*f*) *ta* ⇒ ('*q2*,'*f*) *ta* ⇒ ('*q1* × '*q2*, '*f*) *ta-rule fset* **where**
prod-ta-rules $\mathcal{A}$ $\mathcal{B}$ = {| *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* $\mathcal{A}$)$|^+|$))

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

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

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

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

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

3.1.8 Relabeling tree automaton states to natural numbers

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

definition *map-fset-fset-to-nat* :: (′*a* :: *linorder*) *fset fset* ⇒ ′*a fset* ⇒ *nat* **where**

map-fset-fset-to-nat $X = (\lambda x.$ *the* (*mem-idx* (*sorted-list-of-fset* x) (*sorted-list-of-fset* (*sorted-list-of-fset* |`| X))))

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

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

— The instantiation of < and ≤ for finite sets are |⊂| and |⊆| which don't give rise to a total order and therefore it cannot be an instance of the type class linorder. However taking the lexographic order of the sorted list of each finite set gives rise to a total order. Therefore we provide a relabeling for tree automata where the states are finite sets. This allows us to relabel the well known power set construction.

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

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

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

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

lemma *rules-transfer* [*transfer-rule*]:
rel-fun (=) (*pcr-fset* (=)) *srules rules* ⟨*proof*⟩

lemma *eps-transfer* [*transfer-rule*]:
rel-fun (=) (*pcr-fset* (=)) *seps eps* ⟨*proof*⟩

lemma *TA-equalityI*:
rules $\mathcal{A}$ = *rules* $\mathcal{B}$ ⟹ *eps* $\mathcal{A}$ = *eps* $\mathcal{B}$ ⟹ $\mathcal{A} = \mathcal{B}$
⟨*proof*⟩

lemma *rule-states-code* [*code*]:
rule-states Δ = |⋃| ((λ r. *finsert* (*r-rhs* r) (*fset-of-list* (*r-lhs-states* r))) |`| Δ)
⟨*proof*⟩

lemma *eps-states-code* [*code*]:
eps-states Δ_ε = |⋃| ((λ (q,q'). {|q,q'|}) |`| Δ_ε) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

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

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

lemma *rule-states-union* [*simp*]:
rule-states (Δ |∪| Γ) = *rule-states* Δ |∪| *rule-states* Γ
⟨*proof*⟩

lemma *rule-states-mono*:
Δ |⊆| Γ ⟹ *rule-states* Δ |⊆| *rule-states* Γ
⟨*proof*⟩

lemma *eps-states-union* [*simp*]:
eps-states (Δ |∪| Γ) = *eps-states* Δ |∪| *eps-states* Γ
⟨*proof*⟩

lemma *eps-states-image* [*simp*]:
eps-states (*map-both* *f* |‘| Δ_ε) = *f* |‘| *eps-states* Δ_ε
⟨*proof*⟩

lemma *eps-states-mono*:
Δ |⊆| Γ ⟹ *eps-states* Δ |⊆| *eps-states* Γ
⟨*proof*⟩

lemma *eps-statesI* [*intro*]:
(*p*, *q*) |∈| Δ ⟹ *p* |∈| *eps-states* Δ
(*p*, *q*) |∈| Δ ⟹ *q* |∈| *eps-states* Δ
⟨*proof*⟩

lemma *eps-statesE* [*elim*]:
assumes *p* |∈| *eps-states* Δ
obtains *q* **where** (*p*, *q*) |∈| Δ ∨ (*q*, *p*) |∈| Δ ⟨*proof*⟩

lemma *rule-statesE* [*elim*]:
assumes *q* |∈| *rule-states* Δ
obtains *f ps p* **where** *TA-rule f ps p* |∈| Δ *q* |∈| (*fset-of-list ps*) ∨ *q* = *p* ⟨*proof*⟩

lemma *rule-statesI* [*intro*]:
assumes *r* |∈| Δ *q* |∈| *finsert* (*r-rhs r*) (*fset-of-list* (*r-lhs-states r*))
shows *q* |∈| *rule-states* Δ ⟨*proof*⟩

Destruction rule for states

lemma *rule-statesD*:
r |∈| (*rules* $\mathcal{A}$) ⟹ *r-rhs r* |∈| $\mathcal{Q}$ $\mathcal{A}$ *f qs* → *q* |∈| (*rules* $\mathcal{A}$) ⟹ *q* |∈| $\mathcal{Q}$ $\mathcal{A}$

r |∈| (*rules* $\mathcal{A}$) $\Longrightarrow$ p |∈| *fset-of-list* (*r-lhs-states* r) $\Longrightarrow$ p |∈| $\mathcal{Q}$ $\mathcal{A}$
f *qs* $\to$ q |∈| (*rules* $\mathcal{A}$) $\Longrightarrow$ p |∈| *fset-of-list qs* $\Longrightarrow$ p |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *eps-states* [*simp*]: (*eps* $\mathcal{A}$) |⊆| $\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *eps-statesD*: (p, q) |∈| (*eps* $\mathcal{A}$) $\Longrightarrow$ p |∈| $\mathcal{Q}$ $\mathcal{A}$ ∧ q |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *eps-trancl-statesD*:
(p, q) |∈| (*eps* $\mathcal{A}$)|$^+$| $\Longrightarrow$ p |∈| $\mathcal{Q}$ $\mathcal{A}$ ∧ q |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

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

Mapping over function symbols/states

lemma *finite-Collect-ta-rule*:
finite {*TA-rule f qs q* | *f qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$} (**is** *finite ?S*)
⟨*proof*⟩

lemma *map-ta-rule-finite*:
finite Δ $\Longrightarrow$ *finite* {*TA-rule* (*g h*) (*map f qs*) (*f q*) | *h qs q*. *TA-rule h qs q* ∈ Δ}
⟨*proof*⟩

lemmas *map-ta-rule-fset-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ **for** Δ, *simplified*, *unfolded fmember.rep-eq*[*symmetric*]]
lemmas *map-ta-rule-states-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ *id* **for** Δ, *simplified*, *unfolded fmember.rep-eq*[*symmetric*]]
lemmas *map-ta-rule-funsym-finite* [*simp*] = *map-ta-rule-finite*[*of fset* Δ - *id* **for** Δ, *simplified*, *unfolded fmember.rep-eq*[*symmetric*]]

lemma *map-ta-rule-comp*:
map-ta-rule f g ∘ *map-ta-rule f′ g′* = *map-ta-rule* (*f* ∘ *f′*) (*g* ∘ *g′*)
⟨*proof*⟩

lemma *map-ta-rule-cases*:
map-ta-rule f g r = *TA-rule* (*g* (*r-root r*)) (*map f* (*r-lhs-states r*)) (*f* (*r-rhs r*))
⟨*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*
⟨*proof*⟩

lemma *rule-states-image* [*simp*]:
rule-states (*map-ta-rule f g* |\`| Δ) = *f* |\`| *rule-states* Δ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *Q-mono*:
$(rules\ \mathcal{A})\ |\subseteq|\ (rules\ \mathcal{B}) \Longrightarrow (eps\ \mathcal{A})\ |\subseteq|\ (eps\ \mathcal{B}) \Longrightarrow \mathcal{Q}\ \mathcal{A}\ |\subseteq|\ \mathcal{Q}\ \mathcal{B}$
⟨*proof*⟩

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

lemma *finite-states*:
$finite\ \{q.\ \exists\ f\ p\ ps.\ f\ ps \to p\ |\in|\ rules\ \mathcal{A} \wedge (p = q \vee (p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+|)\}$ (**is** *finite ?set*)
⟨*proof*⟩

Collecting all states reachable from target of rules

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

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

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

lemma *ta-sig-mono*:
$(rules\ \mathcal{A})\ |\subseteq|\ (rules\ \mathcal{B}) \Longrightarrow ta\text{-}sig\ \mathcal{A}\ |\subseteq|\ ta\text{-}sig\ \mathcal{B}$
⟨*proof*⟩

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

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

lemma *ta-der-Var*:
$q\ |\in|\ ta\text{-}der\ \mathcal{A}\ (Var\ x) \longleftrightarrow x = q \vee (x,\ q)\ |\in|\ (eps\ \mathcal{A})|^+|$
⟨*proof*⟩

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

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

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

lemma *ta-der′-Var*:
Var q |∈| *ta-der′* $\mathcal{A}$ (*Var x*) ⟷ $x = q \lor (x, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$|
⟨*proof*⟩

lemma *ta-der′-Fun*:
Var q |∈| *ta-der′* $\mathcal{A}$ (*Fun f ts*) ⟷ *q* |∈| *ta-der* $\mathcal{A}$ (*Fun f ts*)
⟨*proof*⟩

lemma *ta-der′-Fun2*:
Fun f ps |∈| *ta-der′* $\mathcal{A}$ (*Fun g ts*) ⟷ $f = g$ ∧ *length ps* = *length ts* ∧ (∀ *i*<*length ts*. *ps* ! *i* |∈| *ta-der′* $\mathcal{A}$ (*ts* ! *i*))
⟨*proof*⟩

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

Induction schemes for the most used cases

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

lemma *ta-der-gterm-induct*[*consumes 1*, *case-names GFun*]:
assumes *reach*: *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
and *Fun*: ⋀*f ts ps p q*. *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ ⟹ *length ts* = *length ps* ⟹ $p = q \lor (p, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹
(⋀*i*. *i* < *length ts* ⟹ *ps* ! *i* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*ts* ! *i*))) ⟹
(⋀*i*. *i* < *length ts* ⟹ *P* (*ts* ! *i*) (*ps* ! *i*)) ⟹ *P* (*GFun f ts*) *q*
shows *P t q* ⟨*proof*⟩

lemma *ta-der-rule-empty*:
assumes *q* |∈| *ta-der* (*TA* {||} Δ_ε) *t*
obtains *p* **where** *t* = *Var p* $p = q \lor (p, q)$ |∈| Δ_ε|$^+$|
⟨*proof*⟩

lemma *ta-der-eps*:
assumes (*p*, *q*) |∈| (*eps* $\mathcal{A}$) **and** *p* |∈| *ta-der* $\mathcal{A}$ *t*
shows *q* |∈| *ta-der* $\mathcal{A}$ *t* ⟨*proof*⟩

lemma *ta-der-trancl-eps*:
assumes (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| **and** *p* |∈| *ta-der* $\mathcal{A}$ *t*

shows q |∈| *ta-der* $\mathcal{A}$ t ⟨*proof*⟩

lemma *ta-der-mono*:
($\mathit{rules}\ \mathcal{A}$) |⊆| ($\mathit{rules}\ \mathcal{B}$) $\Longrightarrow$ ($\mathit{eps}\ \mathcal{A}$) |⊆| ($\mathit{eps}\ \mathcal{B}$) $\Longrightarrow$ *ta-der* $\mathcal{A}$ t |⊆| *ta-der* $\mathcal{B}$ t
⟨*proof*⟩

lemma *ta-der-el-mono*:
($\mathit{rules}\ \mathcal{A}$) |⊆| ($\mathit{rules}\ \mathcal{B}$) $\Longrightarrow$ ($\mathit{eps}\ \mathcal{A}$) |⊆| ($\mathit{eps}\ \mathcal{B}$) $\Longrightarrow$ q |∈| *ta-der* $\mathcal{A}$ t $\Longrightarrow$ q |∈| *ta-der* $\mathcal{B}$ t
⟨*proof*⟩

lemma *ta-der′-ta-der*:
assumes t |∈| *ta-der′* $\mathcal{A}$ s p |∈| *ta-der* $\mathcal{A}$ t
shows p |∈| *ta-der* $\mathcal{A}$ s ⟨*proof*⟩

lemma *ta-der′-empty*:
assumes t |∈| *ta-der′* (*TA* {||} {||}) s
shows $t = s$ ⟨*proof*⟩

lemma *ta-der′-to-ta-der*:
Var q |∈| *ta-der′* $\mathcal{A}$ s $\Longrightarrow$ q |∈| *ta-der* $\mathcal{A}$ s
⟨*proof*⟩

lemma *ta-der-to-ta-der′*:
q |∈| *ta-der* $\mathcal{A}$ s $\longleftrightarrow$ *Var* q |∈| *ta-der′* $\mathcal{A}$ s
⟨*proof*⟩

lemma *ta-der′-poss*:
assumes t |∈| *ta-der′* $\mathcal{A}$ s
shows *poss* $t \subseteq$ *poss* s ⟨*proof*⟩

lemma *ta-der′-refl*[*simp*]: t |∈| *ta-der′* $\mathcal{A}$ t
⟨*proof*⟩

lemma *ta-der′-eps*:
assumes *Var* p |∈| *ta-der′* $\mathcal{A}$ s **and** (p, q) |∈| ($\mathit{eps}\ \mathcal{A}$)|$^+$|
shows *Var* q |∈| *ta-der′* $\mathcal{A}$ s ⟨*proof*⟩

lemma *ta-der′-trans*:
assumes t |∈| *ta-der′* $\mathcal{A}$ s **and** u |∈| *ta-der′* $\mathcal{A}$ t
shows u |∈| *ta-der′* $\mathcal{A}$ s ⟨*proof*⟩

Connecting contexts to derivation definition

lemma *ta-der-ctxt*:
assumes p: p |∈| *ta-der* $\mathcal{A}$ t q |∈| *ta-der* $\mathcal{A}$ $C\langle \mathit{Var}\ p\rangle$
shows q |∈| *ta-der* $\mathcal{A}$ $C\langle t\rangle$ ⟨*proof*⟩

lemma *ta-der-eps-ctxt*:
assumes p |∈| *ta-der* A $C\langle \mathit{Var}\ q'\rangle$ **and** (q, q') |∈| ($\mathit{eps}\ A$)|$^+$|

shows p |∈| *ta-der* A $C\langle$ *Var* $q\rangle$
⟨*proof*⟩

lemma *rule-reachable-ctxt-exist*:
assumes *rule*: f *qs* $\to$ q |∈| *rules* $\mathcal{A}$ **and** $i <$ *length qs*
shows $\exists$ C. q |∈| *ta-der* $\mathcal{A}$ (C ⟨*Var* (*qs* ! i)⟩) ⟨*proof*⟩

lemma *ta-der-ctxt-decompose*:
assumes q |∈| *ta-der* $\mathcal{A}$ $C\langle t\rangle$
shows $\exists$ p . p |∈| *ta-der* $\mathcal{A}$ t $\wedge$ q |∈| *ta-der* $\mathcal{A}$ $C\langle$ *Var* $p\rangle$ ⟨*proof*⟩

lemma *ta-der-states*:
ta-der $\mathcal{A}$ t |⊆| $\mathcal{Q}$ $\mathcal{A}$ |∪| *fvars-term* t
⟨*proof*⟩

lemma *ground-ta-der-states*:
ground $t \Longrightarrow$ *ta-der* $\mathcal{A}$ t |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

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

lemma *gterm-ta-der-states* [*simp*]:
q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* t) $\Longrightarrow$ q |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-states'*:
q |∈| *ta-der* $\mathcal{A}$ $t \Longrightarrow q$ |∈| $\mathcal{Q}$ $\mathcal{A} \Longrightarrow$ *fvars-term* t |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-not-stateD*:
q |∈| *ta-der* $\mathcal{A}$ $t \Longrightarrow q$ |∉| $\mathcal{Q}$ $\mathcal{A} \Longrightarrow t =$ *Var* q
⟨*proof*⟩

lemma *ta-der-is-fun-stateD*:
is-Fun $t \Longrightarrow q$ |∈| *ta-der* $\mathcal{A}$ $t \Longrightarrow q$ |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-is-fun-fvars-stateD*:
is-Fun $t \Longrightarrow q$ |∈| *ta-der* $\mathcal{A}$ $t \Longrightarrow$ *fvars-term* t |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-not-reach*:
assumes $\bigwedge$ r. r |∈| *rules* $\mathcal{A} \Longrightarrow$ *r-rhs* $r \neq q$
and $\bigwedge$ e. e |∈| *eps* $\mathcal{A} \Longrightarrow$ *snd* $e \neq q$
shows q |∉| *ta-der* $\mathcal{A}$ (*term-of-gterm* t) ⟨*proof*⟩

lemma *ta-rhs-states-subset-states*: *ta-rhs-states* $\mathcal{A}$ |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

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

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

lemma *ta-der-adapt-vars-ground* [*simp*]:
ground t $\Longrightarrow$ *ta-der A* (*adapt-vars t*) = *ta-der A t*
⟨*proof*⟩

lemma *gterm-of-term-inv'*:
ground t $\Longrightarrow$ *term-of-gterm* (*gterm-of-term t*) = *adapt-vars t*
⟨*proof*⟩

lemma *map-vars-term-term-of-gterm*:
map-vars-term f (*term-of-gterm t*) = *term-of-gterm t*
⟨*proof*⟩

lemma *adapt-vars-term-of-gterm*:
adapt-vars (*term-of-gterm t*) = *term-of-gterm t*
⟨*proof*⟩

lemma *ta-der-term-sig*:
q $|\in|$ *ta-der* $\mathcal{A}$ *t* $\Longrightarrow$ *ffunas-term t* $|\subseteq|$ *ta-sig* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-gterm-sig*:
q $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) $\Longrightarrow$ *ffunas-gterm t* $|\subseteq|$ *ta-sig* $\mathcal{A}$
⟨*proof*⟩

ta-lang for terms with arbitrary variable type

lemma *ta-langE*: **assumes** $t \in$ *ta-lang Q* $\mathcal{A}$
obtains t' *q* **where** *ground* t' *q* $|\in|$ *Q q* $|\in|$ *ta-der* $\mathcal{A}$ t' t = *adapt-vars* t'
⟨*proof*⟩

lemma *ta-langI*: **assumes** *ground* t' *q* $|\in|$ *Q q* $|\in|$ *ta-der* $\mathcal{A}$ t' t = *adapt-vars* t'
shows $t \in$ *ta-lang Q* $\mathcal{A}$
⟨*proof*⟩

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

ta-lang for *gterms*

lemma *ta-lang-to-gta-lang* [*simp*]:
ta-lang Q $\mathcal{A}$ = *term-of-gterm* ‘ *gta-lang Q* $\mathcal{A}$ (**is** *?Ls* = *?Rs*)

⟨*proof*⟩

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

lemma *gta-lang-def-sym*:
gterm-of-term ‘ *ta-lang* $Q\ \mathcal{A}$ = *gta-lang* $Q\ \mathcal{A}$

⟨*proof*⟩

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

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

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

lemma *gta-lang-term-of-gterm* [*simp*]:
term-of-gterm $t \in$ *term-of-gterm* ‘ *gta-lang* $Q\ \mathcal{A} \longleftrightarrow t \in$ *gta-lang* $Q\ \mathcal{A}$
⟨*proof*⟩

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

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

lemma *ta-syms-lang*: $t \in$ *ta-lang* $Q\ \mathcal{A} \Longrightarrow$ *ffunas-term* t |⊆| *ta-sig* $\mathcal{A}$
⟨*proof*⟩

lemma *gta-lang-Rest-states-conv*:
gta-lang $Q\ \mathcal{A}$ = *gta-lang* (Q |∩| $\mathcal{Q}\ \mathcal{A}$) $\mathcal{A}$
⟨*proof*⟩

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

Deterministic tree automatons

definition *ta-det* :: $('q,'f)$ *ta* ⇒ *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 $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \longleftrightarrow rules\ \mathcal{A}\ |\subseteq|\ rules\ \mathcal{B} \wedge eps\ \mathcal{A}\ |\subseteq|\ eps\ \mathcal{B}$

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

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

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

lemma *ta-subset-trans*: $ta\text{-}subset\ \mathcal{A}\ \mathcal{B} \Longrightarrow ta\text{-}subset\ \mathcal{B}\ \mathfrak{C} \Longrightarrow ta\text{-}subset\ \mathcal{A}\ \mathfrak{C}$
⟨*proof*⟩

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

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

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

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

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

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

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*:
$fvars\text{-}term\ t\ |\subseteq|\ Q \Longrightarrow q\ |\in|\ ta\text{-}der\ (ta\text{-}restrict\ \mathcal{A}\ Q)\ t \Longrightarrow q\ |\in|\ Q$
⟨*proof*⟩

lemma *ta-derict-ruleI* [*intro*]:

TA-rule f qs q |∈| *rules* $\mathcal{A}$ ⟹ *fset-of-list qs* |⊆| *Q* ⟹ *q* |∈| *Q* ⟹ *TA-rule f qs q* |∈| *rules* (*ta-restrict* $\mathcal{A}$ *Q*)
⟨*proof*⟩

Reachable and productive states: There always is a trim automaton

lemma *finite-ta-reachable* [*simp*]:
finite {*q*. ∃ *t*. *ground t* ∧ *q* |∈| *ta-der* $\mathcal{A}$ *t*}
⟨*proof*⟩

lemma *ta-reachable-states*:
ta-reachable $\mathcal{A}$ |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-reachableE*:
assumes *q* |∈| *ta-reachable* $\mathcal{A}$
obtains *t* **where** *ground t q* |∈| *ta-der* $\mathcal{A}$ *t*
⟨*proof*⟩

lemma *ta-reachable-gtermE* [*elim*]:
assumes *q* |∈| *ta-reachable* $\mathcal{A}$
obtains *t* **where** *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
⟨*proof*⟩

lemma *ta-reachableI* [*intro*]:
assumes *ground t* **and** *q* |∈| *ta-der* $\mathcal{A}$ *t*
shows *q* |∈| *ta-reachable* $\mathcal{A}$
⟨*proof*⟩

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

lemma *ta-reachableI-rule*:
assumes *sub*: *fset-of-list qs* |⊆| *ta-reachable* $\mathcal{A}$
and *rule*: *TA-rule f qs q* |∈| *rules* $\mathcal{A}$
shows *q* |∈| *ta-reachable* $\mathcal{A}$
∃ *ts*. *length qs* = *length ts* ∧ (∀ *i* < *length ts*. *ground* (*ts* ! *i*)) ∧
(∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der* $\mathcal{A}$ (*ts* ! *i*)) (**is** *?G*)
⟨*proof*⟩

lemma *ta-reachable-rule-gtermE*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
and *TA-rule f qs q* |∈| *rules* $\mathcal{A}$
obtains *t* **where** *groot t* = (*f*, *length qs*) *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
⟨*proof*⟩

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

shows q' |∈| *ta-reachable* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-reachableI-eps*:
assumes *reach*: q |∈| *ta-reachable* $\mathcal{A}$
and *eps*: (q, q') |∈| *eps* $\mathcal{A}$
shows q' |∈| *ta-reachable* $\mathcal{A}$
⟨*proof*⟩

lemma *finite-ta-productive*:
finite $\{p.\ \exists\, q\ q'\ C.\ p = q \wedge q'$ |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ q\rangle \wedge q'$ |∈| $P\}$
⟨*proof*⟩

lemma *ta-productiveE*: **assumes** q |∈| *ta-productive* P $\mathcal{A}$
obtains q' C **where** q' |∈| *ta-der* $\mathcal{A}$ $(C\langle Var\ q\rangle)$ q' |∈| P
⟨*proof*⟩

lemma *ta-productiveI*:
assumes q' |∈| *ta-der* $\mathcal{A}$ $(C\langle Var\ q\rangle)$ q' |∈| P
shows q |∈| *ta-productive* P $\mathcal{A}$
⟨*proof*⟩

lemma *ta-productiveI'*:
assumes q |∈| *ta-der* $\mathcal{A}$ $(C\langle Var\ p\rangle)$ q |∈| *ta-productive* P $\mathcal{A}$
shows p |∈| *ta-productive* P $\mathcal{A}$
⟨*proof*⟩

lemma *ta-productive-setI*:
q |∈| $P \Longrightarrow q$ |∈| *ta-productive* P $\mathcal{A}$
⟨*proof*⟩

lemma *ta-reachable-empty-rules* [*simp*]:
rules $\mathcal{A} = \{||\} \Longrightarrow$ *ta-reachable* $\mathcal{A} = \{||\}$
⟨*proof*⟩

lemma *ta-reachable-mono*:
ta-subset $\mathcal{A}$ $\mathcal{B} \Longrightarrow$ *ta-reachable* $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{B}$ ⟨*proof*⟩

lemma *ta-reachabe-rhs-states*:
ta-reachable $\mathcal{A}$ |⊆| *ta-rhs-states* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-reachable-eps*:
(p, q) |∈| $(eps\ \mathcal{A})|^+|$ $\Longrightarrow$ p |∈| *ta-reachable* $\mathcal{A}$ $\Longrightarrow$ (p, q) |∈| $(fRestr\ (eps\ \mathcal{A})$ $(ta\text{-}reachable\ \mathcal{A}))|^+|$
⟨*proof*⟩

lemma *ta-der-only-reach*:
assumes *fvars-term t* |⊆| *ta-reachable* $\mathcal{A}$
shows *ta-der* $\mathcal{A}$ *t* = *ta-der* (*ta-only-reach* $\mathcal{A}$) *t* (**is** *?LS* = *?RS*)
⟨*proof*⟩

lemma *ta-der-gterm-only-reach*:
ta-der $\mathcal{A}$ (*term-of-gterm t*) = *ta-der* (*ta-only-reach* $\mathcal{A}$) (*term-of-gterm t*)
⟨*proof*⟩

lemma *ta-reachable-ta-only-reach* [*simp*]:
ta-reachable (*ta-only-reach* $\mathcal{A}$) = *ta-reachable* $\mathcal{A}$ (**is** *?LS* = *?RS*)
⟨*proof*⟩

lemma *ta-only-reach-reachable*:
$\mathcal{Q}$ (*ta-only-reach* $\mathcal{A}$) |⊆| *ta-reachable* (*ta-only-reach* $\mathcal{A}$)
⟨*proof*⟩

lemma *gta-only-reach-lang*:
gta-lang Q (*ta-only-reach* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
⟨*proof*⟩

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

lemma *ta-only-reach-lang*:
ta-lang Q (*ta-only-reach* $\mathcal{A}$) = *ta-lang Q* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-prod-epsD*:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ *p* |∈| *ta-productive P* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-only-prod-eps*:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)|$^+$| ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ (*p*, *q*) |∈| (*eps* (*ta-only-prod* *P* $\mathcal{A}$))|$^+$|
⟨*proof*⟩

lemma *ta-der-only-prod*:
q |∈| *ta-der* $\mathcal{A}$ *t* ⟹ *q* |∈| *ta-productive P* $\mathcal{A}$ ⟹ *q* |∈| *ta-der* (*ta-only-prod P* $\mathcal{A}$) *t*
⟨*proof*⟩

lemma *ta-der-ta-only-prod-ta-der*:
q |∈| *ta-der* (*ta-only-prod P* $\mathcal{A}$) *t* ⟹ *q* |∈| *ta-der* $\mathcal{A}$ *t*

⟨*proof*⟩

lemma *gta-only-prod-lang*:
gta-lang Q (*ta-only-prod Q* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$ (**is** *gta-lang Q ?*$\mathcal{A}$ = -)
⟨*proof*⟩

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

lemma *ta-only-prod-lang*:
ta-lang Q (*ta-only-prod Q* $\mathcal{A}$) = *ta-lang Q* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-prodictive-ta-only-prod* [*simp*]:
ta-productive P (*ta-only-prod P* $\mathcal{A}$) = *ta-productive P* $\mathcal{A}$ (**is** *?LS* = *?RS*)
⟨*proof*⟩

lemma *ta-only-prod-productive*:
$\mathcal{Q}$ (*ta-only-prod P* $\mathcal{A}$) |⊆| *ta-productive P* (*ta-only-prod P* $\mathcal{A}$)
⟨*proof*⟩

lemma *ta-only-prod-reachable*:
assumes *all-reach*: $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows $\mathcal{Q}$ (*ta-only-prod P* $\mathcal{A}$) |⊆| *ta-reachable* (*ta-only-prod P* $\mathcal{A}$) (**is** *?Ls* |⊆| *?Rs*)
⟨*proof*⟩

lemma *ta-prod-reach-subset*:
ta-subset (*ta-only-prod P* (*ta-only-reach* $\mathcal{A}$)) $\mathcal{A}$
⟨*proof*⟩

lemma *ta-prod-reach-states*:
$\mathcal{Q}$ (*ta-only-prod P* (*ta-only-reach* $\mathcal{A}$)) |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *ta-productive-aux*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$ *q* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*t*⟩)
shows ∃ *C'*. *ground-ctxt C'* ∧ *q* |∈| *ta-der* $\mathcal{A}$ (*C'*⟨*t*⟩) ⟨*proof*⟩

lemma *ta-productive-def'*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows *ta-productive Q* $\mathcal{A}$ = {| *q*| *q q' C*. *ground-ctxt C* ∧ *q'* |∈| *ta-der* $\mathcal{A}$ (*C*⟨*Var q*⟩) ∧ *q'* |∈| *Q* |}
⟨*proof*⟩

lemma *trim-gta-lang*: *gta-lang Q* (*trim-ta Q* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
⟨*proof*⟩

lemma *trim-ta-subset*: *ta-subset* (*trim-ta Q* $\mathcal{A}$) $\mathcal{A}$
⟨*proof*⟩

theorem *trim-ta*: *ta-is-trim Q* (*trim-ta Q* $\mathcal{A}$) ⟨*proof*⟩

lemma *reg-is-trim-trim-reg* [*simp*]: *reg-is-trim* (*trim-reg R*)
⟨*proof*⟩

lemma *trim-reg-reach* [*simp*]:
$\mathcal{Q}_r$ (*trim-reg A*) |⊆| *ta-reachable* (*ta* (*trim-reg A*))
⟨*proof*⟩

lemma *trim-reg-prod* [*simp*]:
$\mathcal{Q}_r$ (*trim-reg A*) |⊆| *ta-productive* (*fin* (*trim-reg A*)) (*ta* (*trim-reg A*))
⟨*proof*⟩

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

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

Map function over TA rules which change states/signature

lemma *map-ta-rule-iff*:
map-ta-rule f g |'| Δ = {|*TA-rule* (*g h*) (*map f qs*) (*f q*) | *h qs q*. *TA-rule h qs q* |∈| Δ|}
⟨*proof*⟩

lemma $\mathcal{L}$-*trim*: $\mathcal{L}$ (*trim-reg R*) = $\mathcal{L}$ *R*
⟨*proof*⟩

lemma *fmap-funs-ta-def'*:
fmap-funs-ta h $\mathcal{A}$ = *TA* {|(*h f*) *qs* → *q* |*f qs q*. *f qs* → *q* |∈| *rules* $\mathcal{A}$|} (*eps* $\mathcal{A}$)
⟨*proof*⟩

lemma *fmap-states-ta-def'*:
fmap-states-ta h $\mathcal{A}$ = *TA* {|*f* (*map h qs*) → *h q* |*f qs q*. *f qs* → *q* |∈| *rules* $\mathcal{A}$|} (*map-both h* |'| *eps* $\mathcal{A}$)
⟨*proof*⟩

lemma *fmap-states* [*simp*]:
$\mathcal{Q}$ (*fmap-states-ta h* $\mathcal{A}$) = *h* |'| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *fmap-states-ta-sig* [*simp*]:
ta-sig (*fmap-states-ta f* $\mathcal{A}$) = *ta-sig* $\mathcal{A}$
⟨*proof*⟩

lemma *fmap-states-ta-eps-wit*:
assumes (*h p*, *q*) |∈| (*map-both h* |'| *eps* $\mathcal{A}$)|$^+$| *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$) *p* |∈| $\mathcal{Q}$ $\mathcal{A}$
obtains *q′* **where** *q* = *h q′* (*p*, *q′*) |∈| (*eps* $\mathcal{A}$)|$^+$| *q′* |∈| $\mathcal{Q}$ $\mathcal{A}$ ⟨*proof*⟩

lemma *ta-der-fmap-states-inv-superset*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |⊆| $\mathcal{B}$ *finj-on h* $\mathcal{B}$
and *q* |∈| *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*)
shows *the-finv-into* $\mathcal{B}$ *h q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟨*proof*⟩

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

lemma *ta-der-to-fmap-states-der*:
assumes *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
shows *h q* |∈| *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*) ⟨*proof*⟩

lemma *ta-der-fmap-states-conv*:
assumes *finj-on h* ($\mathcal{Q}$ $\mathcal{A}$)
shows *ta-der* (*fmap-states-ta h* $\mathcal{A}$) (*term-of-gterm t*) = *h* |'| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
⟨*proof*⟩

lemma *fmap-states-ta-det*:
assumes *finj-on f* ($\mathcal{Q}$ $\mathcal{A}$)
shows *ta-det* (*fmap-states-ta f* $\mathcal{A}$) = *ta-det* $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *fmap-states-ta-lang*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$) ⟹ *Q* |⊆| $\mathcal{Q}$ $\mathcal{A}$ ⟹ *gta-lang* (*f* |'| *Q*) (*fmap-states-ta f* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
⟨*proof*⟩

lemma *fmap-states-ta-lang2*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$ |∪| *Q*) ⟹ *gta-lang* (*f* |'| *Q*) (*fmap-states-ta f* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$
⟨*proof*⟩

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

lemma *funs-ta*[*code*]:
funs-ta $\mathcal{A}$ = (λ *r*. *case r of TA-rule f ps p* ⇒ *f*) |'| (*rules* $\mathcal{A}$) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *finite-funs-ta* [*simp*]:
finite {*f*. ∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$}
⟨*proof*⟩

lemma *funs-taE* [*elim*]:
assumes *f* |∈| *funs-ta* $\mathcal{A}$
obtains *ps p* **where** *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ ⟨*proof*⟩

lemma *funs-taI* [*intro*]:
TA-rule f ps p |∈| *rules* $\mathcal{A}$ ⟹ *f* |∈| *funs-ta* $\mathcal{A}$
⟨*proof*⟩

lemma *fmap-funs-ta-cong*:
(⋀*x*. *x* |∈| *funs-ta* $\mathcal{A}$ ⟹ *h x* = *k x*) ⟹ $\mathcal{A}$ = $\mathcal{B}$ ⟹ *fmap-funs-ta h* $\mathcal{A}$ = *fmap-funs-ta k* $\mathcal{B}$
⟨*proof*⟩

lemma [*simp*]: {|*TA-rule f qs q* |*f qs q*. *TA-rule f qs q* |∈| *X*|} = *X*
⟨*proof*⟩

lemma *fmap-funs-ta-id* [*simp*]:
fmap-funs-ta id $\mathcal{A}$ = $\mathcal{A}$ ⟨*proof*⟩

lemma *fmap-states-ta-id* [*simp*]:
fmap-states-ta id $\mathcal{A}$ = $\mathcal{A}$
⟨*proof*⟩

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

lemma *fmap-funs-ta-comp*:
fmap-funs-ta h (*fmap-funs-ta k A*) = *fmap-funs-ta* (*h* ∘ *k*) *A*
⟨*proof*⟩

lemma *fmap-funs-reg-comp*:
fmap-funs-reg h (*fmap-funs-reg k A*) = *fmap-funs-reg* (*h* ∘ *k*) *A*
⟨*proof*⟩

lemma *fmap-states-ta-comp*:
fmap-states-ta h (*fmap-states-ta k A*) = *fmap-states-ta* (*h* ∘ *k*) *A*
⟨*proof*⟩

lemma *funs-ta-fmap-funs-ta* [*simp*]:
funs-ta (*fmap-funs-ta f A*) = *f* |'| *funs-ta A*
⟨*proof*⟩

lemma *ta-der-funs-ta*:
q |∈| *ta-der* A $t \Longrightarrow$ *ffuns-term* t |⊆| *funs-ta* A
⟨*proof*⟩

lemma *ta-der-fmap-funs-ta*:
q |∈| *ta-der* A $t \Longrightarrow q$ |∈| *ta-der* (*fmap-funs-ta* f A) (*map-funs-term* f t)
⟨*proof*⟩

lemma *ta-der-fmap-states-ta*:
assumes q |∈| *ta-der* A t
shows h q |∈| *ta-der* (*fmap-states-ta* h A) (*map-vars-term* h t)
⟨*proof*⟩

lemma *ta-der-fmap-states-ta-mono*:
shows f |'| *ta-der* A (*term-of-gterm* s) |⊆| *ta-der* (*fmap-states-ta* f A) (*term-of-gterm* s)
⟨*proof*⟩

lemma *ta-der-fmap-states-ta-mono2*:
assumes *finj-on* f ($\mathcal{Q}$ A)
shows *ta-der* (*fmap-states-ta* f A) (*term-of-gterm* s) |⊆| f |'| *ta-der* A (*term-of-gterm* s)
⟨*proof*⟩

lemma *fmap-funs-ta-der′*:
q |∈| *ta-der* (*fmap-funs-ta* h A) $t \Longrightarrow \exists\, t'.$ q |∈| *ta-der* A $t' \wedge$ *map-funs-term* h $t' = t$
⟨*proof*⟩

lemma *fmap-funs-gta-lang*:
gta-lang Q (*fmap-funs-ta* h $\mathcal{A}$) = *map-gterm* h ' *gta-lang* Q $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *fmap-funs-*$\mathcal{L}$:
$\mathcal{L}$ (*fmap-funs-reg* h R) = *map-gterm* h ' $\mathcal{L}$ R
⟨*proof*⟩

lemma *ta-states-fmap-funs-ta* [*simp*]: $\mathcal{Q}$ (*fmap-funs-ta* f A) = $\mathcal{Q}$ A
⟨*proof*⟩

lemma *ta-reachable-fmap-funs-ta* [*simp*]:
ta-reachable (*fmap-funs-ta* f A) = *ta-reachable* A ⟨*proof*⟩

lemma *fin-in-states*:
fin (*reg-Restr-*Q_f R) |⊆| $\mathcal{Q}_r$ (*reg-Restr-*Q_f R)
⟨*proof*⟩

lemma *fmap-states-reg-Restr-*Q_f*-fin*:
finj-on f ($\mathcal{Q}$ $\mathcal{A}$) $\Longrightarrow$ *fin* (*fmap-states-reg f* (*reg-Restr-*Q_f *R*)) |⊆| $\mathcal{Q}_r$ (*fmap-states-reg f* (*reg-Restr-*Q_f *R*))
⟨*proof*⟩

lemma $\mathcal{L}$*-fmap-states-reg-Inl-Inr* [*simp*]:
$\mathcal{L}$ (*fmap-states-reg Inl R*) = $\mathcal{L}$ *R*
$\mathcal{L}$ (*fmap-states-reg Inr R*) = $\mathcal{L}$ *R*
⟨*proof*⟩

lemma *finite-Collect-prod-ta-rules*:
finite {*f qs* → (*a*, *b*) |*f qs a b*. *f map fst qs* → *a* |∈| *rules* $\mathcal{A}$ ∧ *f map snd qs* → *b* |∈| *rules* $\mathfrak{B}$} (**is** *finite ?set*)
⟨*proof*⟩

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

lemma *finite-prod-epsLp*:
finite (*Collect* (*prod-epsLp* $\mathcal{A}$ $\mathcal{B}$))
⟨*proof*⟩

lemma *finite-prod-epsRp*:
finite (*Collect* (*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 qs* → *q* |∈| *rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) ⟷ *f qs* → *q* |∈| *prod-ta-rules* $\mathcal{A}$ $\mathcal{B}$
r |∈| *rules* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) ⟷ *r* |∈| *prod-ta-rules* $\mathcal{A}$ $\mathcal{B}$
⟨*proof*⟩

lemma *prod-ta-states*:
$\mathcal{Q}$ (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) |⊆| $\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$
⟨*proof*⟩

lemma *prod-ta-det*:
assumes *ta-det* $\mathcal{A}$ **and** *ta-det* $\mathcal{B}$
shows *ta-det* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$)
⟨*proof*⟩

lemma *prod-ta-sig*:
ta-sig (*prod-ta* $\mathcal{A}$ $\mathcal{B}$) |⊆| *ta-sig* $\mathcal{A}$ |∪| *ta-sig* $\mathcal{B}$
⟨*proof*⟩

lemma *from-prod-eps*:
(*p*, *q*) |∈| (*eps* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$))|$^+$| $\Longrightarrow$ (*snd p*, *snd q*) |∉| (*eps* $\mathcal{B}$)|$^+$| $\Longrightarrow$ *snd p* = *snd q* ∧ (*fst p*, *fst q*) |∈| (*eps* $\mathcal{A}$)|$^+$|
(*p*, *q*) |∈| (*eps* (*prod-ta* $\mathcal{A}$ $\mathcal{B}$))|$^+$| $\Longrightarrow$ (*fst p*, *fst q*) |∉| (*eps* $\mathcal{A}$)|$^+$| $\Longrightarrow$ *fst p* = *fst*

$q \wedge (snd\ p,\ snd\ q)\ |\in|\ (eps\ \mathcal{B})|^+|$
  ⟨*proof*⟩

lemma *to-prod-eps𝒜*:
  $(p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+| \Longrightarrow r\ |\in|\ \mathcal{Q}\ \mathcal{B} \Longrightarrow ((p,\ r),\ (q,\ r))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
  ⟨*proof*⟩

lemma *to-prod-eps𝓑*:
  $(p,\ q)\ |\in|\ (eps\ \mathcal{B})|^+| \Longrightarrow r\ |\in|\ \mathcal{Q}\ \mathcal{A} \Longrightarrow ((r,\ p),\ (r,\ q))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
  ⟨*proof*⟩

lemma *to-prod-eps*:
  $(p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+| \Longrightarrow (p',\ q')\ |\in|\ (eps\ \mathcal{B})|^+| \Longrightarrow ((p,\ p'),\ (q,\ q'))\ |\in|\ (eps\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}))|^+|$
⟨*proof*⟩

lemma *prod-ta-der-to-𝒜-𝓑-der1*:
  assumes $q\ |\in|\ ta\text{-}der\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B})\ (term\text{-}of\text{-}gterm\ t)$
  shows $fst\ q\ |\in|\ ta\text{-}der\ \mathcal{A}\ (term\text{-}of\text{-}gterm\ t)$ ⟨*proof*⟩

lemma *prod-ta-der-to-𝒜-𝓑-der2*:
  assumes $q\ |\in|\ ta\text{-}der\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B})\ (term\text{-}of\text{-}gterm\ t)$
  shows $snd\ q\ |\in|\ ta\text{-}der\ \mathcal{B}\ (term\text{-}of\text{-}gterm\ t)$ ⟨*proof*⟩

lemma *𝒜-𝓑-der-to-prod-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 $q\ |\in|\ ta\text{-}der\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B})\ (term\text{-}of\text{-}gterm\ t)$ ⟨*proof*⟩

lemma *prod-ta-der*:
  $q\ |\in|\ ta\text{-}der\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B})\ (term\text{-}of\text{-}gterm\ t) \longleftrightarrow$
    $fst\ q\ |\in|\ ta\text{-}der\ \mathcal{A}\ (term\text{-}of\text{-}gterm\ t) \wedge snd\ q\ |\in|\ ta\text{-}der\ \mathcal{B}\ (term\text{-}of\text{-}gterm\ t)$
  ⟨*proof*⟩

lemma *intersect-ta-gta-lang*:
 $gta\text{-}lang\ (Q_{\mathcal{A}}\ |\times|\ Q_{\mathcal{B}})\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}) = gta\text{-}lang\ Q_{\mathcal{A}}\ \mathcal{A} \cap gta\text{-}lang\ Q_{\mathcal{B}}\ \mathcal{B}$
  ⟨*proof*⟩

lemma *ℒ-intersect*: $\mathcal{L}\ (reg\text{-}intersect\ R\ L) = \mathcal{L}\ R \cap \mathcal{L}\ L$
  ⟨*proof*⟩

lemma *intersect-ta-ta-lang*:
 $ta\text{-}lang\ (Q_{\mathcal{A}}\ |\times|\ Q_{\mathcal{B}})\ (prod\text{-}ta\ \mathcal{A}\ \mathcal{B}) = ta\text{-}lang\ Q_{\mathcal{A}}\ \mathcal{A} \cap ta\text{-}lang\ Q_{\mathcal{B}}\ \mathcal{B}$
  ⟨*proof*⟩

lemma *ta-union-ta-subset*:
  $ta\text{-}subset\ \mathcal{A}\ (ta\text{-}union\ \mathcal{A}\ \mathcal{B})\ ta\text{-}subset\ \mathcal{B}\ (ta\text{-}union\ \mathcal{A}\ \mathcal{B})$
  ⟨*proof*⟩

lemma *ta-union-states* [*simp*]:
$\mathcal{Q}$ (*ta-union* $\mathcal{A}$ $\mathcal{B}$) = $\mathcal{Q}$ $\mathcal{A}$ |∪| $\mathcal{Q}$ $\mathcal{B}$
⟨*proof*⟩

lemma *ta-union-sig* [*simp*]:
ta-sig (*ta-union* $\mathcal{A}$ $\mathcal{B}$) = *ta-sig* $\mathcal{A}$ |∪| *ta-sig* $\mathcal{B}$
⟨*proof*⟩

lemma *ta-union-eps-disj-states*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} **and** (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))$|^+|$
shows (*p*, *q*) |∈| (*eps* $\mathcal{A}$)$|^+|$ ∨ (*p*, *q*) |∈| (*eps* $\mathcal{B}$)$|^+|$ ⟨*proof*⟩

lemma *eps-ta-union-eps* [*simp*]:
(*p*, *q*) |∈| (*eps* $\mathcal{A}$)$|^+|$ ⟹ (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))$|^+|$
(*p*, *q*) |∈| (*eps* $\mathcal{B}$)$|^+|$ ⟹ (*p*, *q*) |∈| (*eps* (*ta-union* $\mathcal{A}$ $\mathcal{B}$))$|^+|$
⟨*proof*⟩

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

lemma *ta-union-der-disj-states*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} **and** *q* |∈| *ta-der* (*ta-union* $\mathcal{A}$ $\mathcal{B}$) *t*
shows *q* |∈| *ta-der* $\mathcal{A}$ *t* ∨ *q* |∈| *ta-der* $\mathcal{B}$ *t* ⟨*proof*⟩

lemma *ta-union-der-disj-states′*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||}
shows *ta-der* (*ta-union* $\mathcal{A}$ $\mathcal{B}$) *t* = *ta-der* $\mathcal{A}$ *t* |∪| *ta-der* $\mathcal{B}$ *t*
⟨*proof*⟩

lemma *ta-union-gta-lang*:
assumes $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||} **and** $Q_{\mathcal{A}}$ |⊆| $\mathcal{Q}$ $\mathcal{A}$ **and** $Q_{\mathcal{B}}$ |⊆| $\mathcal{Q}$ $\mathcal{B}$
shows*gta-lang* ($Q_{\mathcal{A}}$ |∪| $Q_{\mathcal{B}}$) (*ta-union* $\mathcal{A}$ $\mathcal{B}$) = *gta-lang* $Q_{\mathcal{A}}$ $\mathcal{A}$ ∪ *gta-lang* $Q_{\mathcal{B}}$ $\mathcal{B}$
(**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma $\mathcal{L}$*-union*: $\mathcal{L}$ (*reg-union R L*) = $\mathcal{L}$ *R* ∪ $\mathcal{L}$ *L*
⟨*proof*⟩

lemma *reg-union-states*:
$\mathcal{Q}_r$ (*reg-union A B*) = (*Inl* |'| $\mathcal{Q}_r$ *A*) |∪| (*Inr* |'| $\mathcal{Q}_r$ *B*)
⟨*proof*⟩

lemma *ta-empty* [*simp*]:
ta-empty Q $\mathcal{A}$ = (*gta-lang Q* $\mathcal{A}$ = {})

⟨*proof*⟩

lemma *reg-empty* [*simp*]:
reg-empty $R = (\mathcal{L}\ R = \{\})$
⟨*proof*⟩

Epsilon free automaton

lemma *finite-eps-free-rulep* [*simp*]:
finite (*Collect* (*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*:
ta-der (*eps-free* $\mathcal{A}$) (*term-of-gterm t*) = *ta-der* $\mathcal{A}$ (*term-of-gterm t*) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *ta-lang-eps-free* [*simp*]:
gta-lang Q (*eps-free* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$
⟨*proof*⟩

lemma $\mathcal{L}$*-eps-free*: $\mathcal{L}$ (*eps-free-reg* R) = $\mathcal{L}$ R
⟨*proof*⟩

Sufficient criterion for containment

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

lemma *ta-contains-aux-state-set*:
assumes *ta-contains-aux* $\mathcal{F}$ Q $\mathcal{A}$ Q $t \in \mathcal{T}_G\ \mathcal{F}$
shows $\exists$ q. q $|\in|$ Q $\wedge$ q $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟨*proof*⟩

lemma *ta-contains-aux-mono*:
assumes *ta-subset* $\mathcal{A}$ $\mathcal{B}$ **and** Q_2 $|\subseteq|$ Q_2'
shows *ta-contains-aux* $\mathcal{F}$ Q_1 $\mathcal{A}$ Q_2 $\Longrightarrow$ *ta-contains-aux* $\mathcal{F}$ Q_1 $\mathcal{B}$ Q_2'
⟨*proof*⟩

definition *ta-contains* :: $('f \times nat)$ *set* $\Rightarrow$ $('f \times nat)$ *set* $\Rightarrow$ $('q, 'f)$ *ta* $\Rightarrow$ $'q$ *fset* $\Rightarrow$ $'q$ *fset* $\Rightarrow$ *bool*
where *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{A}$ Q Q_f $\equiv$ *ta-contains-aux* $\mathcal{F}$ Q $\mathcal{A}$ Q $\wedge$ *ta-contains-aux* $\mathcal{G}$ Q $\mathcal{A}$ Q_f

lemma *ta-contains-mono*:

assumes *ta-subset* $\mathcal{A}$ $\mathcal{B}$ **and** Q_f $|\subseteq|$ $Q_f{}'$
shows *ta-contains* $\mathcal{F}$ $\mathcal{G}$ $\mathcal{A}$ Q $Q_f \Longrightarrow$ *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*. *groot* $t \in \mathcal{G} \Longrightarrow \bigcup$ (*funas-gterm* ' *set* (*gargs* *t*)) $\subseteq \mathcal{F} \Longrightarrow t \in$ *gta-lang* Q_f $\mathcal{A}$
⟨*proof*⟩

lemma *ta-contains*:
assumes *contain*: *ta-contains* $\mathcal{F}$ $\mathcal{F}$ $\mathcal{A}$ Q Q_f
shows $\mathcal{T}_G$ $\mathcal{F} \subseteq$ *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 $|\subseteq|$ X
shows *finj-on* (*map-fset-to-nat* X) Y
⟨*proof*⟩

lemma *map-fset-fset-to-nat-inj*:
assumes Y $|\subseteq|$ X
shows *finj-on* (*map-fset-fset-to-nat* X) Y ⟨*proof*⟩

lemma *relabel-gta-lang* [*simp*]:
gta-lang (*relabel-*Q_f Q $\mathcal{A}$) (*relabel-ta* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$
⟨*proof*⟩

lemma $\mathcal{L}$*-relable* [*simp*]: $\mathcal{L}$ (*relabel-reg* R) = $\mathcal{L}$ R
⟨*proof*⟩

lemma *relabel-ta-lang* [*simp*]:
ta-lang (*relabel-*Q_f Q $\mathcal{A}$) (*relabel-ta* $\mathcal{A}$) = *ta-lang* Q $\mathcal{A}$
⟨*proof*⟩

lemma *relabel-fset-gta-lang* [*simp*]:
gta-lang (*relabel-fset-*Q_f Q $\mathcal{A}$) (*relabel-fset-ta* $\mathcal{A}$) = *gta-lang* Q $\mathcal{A}$
⟨*proof*⟩

lemma $\mathcal{L}$*-relable-fset* [*simp*]: $\mathcal{L}$ (*relable-fset-reg* R) = $\mathcal{L}$ R
⟨*proof*⟩

lemma *ta-states-trim-ta*:

$\mathcal{Q}$ (*trim-ta Q* $\mathcal{A}$) |⊆| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *trim-ta-reach*: $\mathcal{Q}$ (*trim-ta Q* $\mathcal{A}$) |⊆| *ta-reachable* (*trim-ta Q* $\mathcal{A}$)
⟨*proof*⟩

lemma *trim-ta-prod*: $\mathcal{Q}$ (*trim-ta Q A*) |⊆| *ta-productive Q* (*trim-ta Q A*)
⟨*proof*⟩

lemma *empty-gta-lang*:
gta-lang Q (*TA* {||} {||}) = {}
⟨*proof*⟩

abbreviation *empty-reg* **where**
empty-reg ≡ *Reg* {||} (*TA* {||} {||})

lemma $\mathcal{L}$-*epmty*:
$\mathcal{L}$ *empty-reg* = {}
⟨*proof*⟩

lemma *const-ta-lang*:
gta-lang {|*q*|} (*TA* {| *TA-rule f* [] *q* |} {||}) = {*GFun f* []}
⟨*proof*⟩

lemma *run-argsD*:
run $\mathcal{A}$ *s t* ⟹ *length* (*gargs s*) = *length* (*gargs t*) ∧ (∀ *i* < *length* (*gargs t*). *run* $\mathcal{A}$ (*gargs s* ! *i*) (*gargs t* ! *i*))
⟨*proof*⟩

lemma *run-root-rule*:
run $\mathcal{A}$ *s t* ⟹ *TA-rule* (*groot-sym t*) (*map ex-comp-state* (*gargs s*)) (*ex-rule-state s*) |∈| (*rules* $\mathcal{A}$) ∧
(*ex-rule-state s* = *ex-comp-state s* ∨ (*ex-rule-state s*, *ex-comp-state s*) |∈| (*eps* $\mathcal{A}$)|$^+$|)
⟨*proof*⟩

lemma *run-poss-eq*: *run* $\mathcal{A}$ *s t* ⟹ *gposs s* = *gposs t*
⟨*proof*⟩

lemma *run-gsubt-cl*:
assumes *run* $\mathcal{A}$ *s t* **and** *p* ∈ *gposs t*
shows *run* $\mathcal{A}$ (*gsubt-at s p*) (*gsubt-at t p*) ⟨*proof*⟩

lemma *run-replace-at*:
assumes *run* $\mathcal{A}$ *s t* **and** *run* $\mathcal{A}$ *u v* **and** *p* ∈ *gposs s*
and *ex-comp-state* (*gsubt-at s p*) = *ex-comp-state u*
shows *run* $\mathcal{A}$ $s[p \leftarrow u]_G$ $t[p \leftarrow v]_G$ ⟨*proof*⟩

relating runs to derivation definition

lemma *run-to-comp-st-gta-der*:
run $\mathcal{A}$ *s t* $\Longrightarrow$ *ex-comp-state s* |∈| *gta-der* $\mathcal{A}$ *t*
⟨*proof*⟩

lemma *run-to-rule-st-gta-der*:
assumes *run* $\mathcal{A}$ *s t* **shows** *ex-rule-state s* |∈| *gta-der* $\mathcal{A}$ *t*
⟨*proof*⟩

lemma *run-to-gta-der-gsubt-at*:
assumes *run* $\mathcal{A}$ *s t* **and** *p* ∈ *gposs t*
shows *ex-rule-state* (*gsubt-at s p*) |∈| *gta-der* $\mathcal{A}$ (*gsubt-at t p*)
ex-comp-state (*gsubt-at s p*) |∈| *gta-der* $\mathcal{A}$ (*gsubt-at t p*)
⟨*proof*⟩

lemma *gta-der-to-run*:
q |∈| *gta-der* $\mathcal{A}$ *t* $\Longrightarrow$ (∃ *p qs*. *run* $\mathcal{A}$ (*GFun* (*p*, *q*) *qs*) *t*) ⟨*proof*⟩

lemma *run-ta-der-ctxt-split1*:
assumes *run* $\mathcal{A}$ *s t p* ∈ *gposs t*
shows *ex-comp-state s* |∈| *ta-der* $\mathcal{A}$ (*ctxt-at-pos* (*term-of-gterm t*) *p*)⟨*Var* (*ex-comp-state* (*gsubt-at s p*))⟩
⟨*proof*⟩

lemma *run-ta-der-ctxt-split2*:
assumes *run* $\mathcal{A}$ *s t p* ∈ *gposs t*
shows *ex-comp-state s* |∈| *ta-der* $\mathcal{A}$ (*ctxt-at-pos* (*term-of-gterm t*) *p*)⟨*Var* (*ex-rule-state* (*gsubt-at s p*))⟩
⟨*proof*⟩

end
theory *Tree-Automata-Det*
imports
Tree-Automata
begin

3.2 Powerset Construction for Tree Automata

The idea to treat states and transitions separately is from arXiv:1511.03595. Some parts of the implementation are also based on that paper. (The Algorithm corresponds roughly to the one in "Step 5")

Abstract Definitions and Correctness Proof

definition *ps-reachable-statesp* **where**
ps-reachable-statesp $\mathcal{A}$ *f ps* = (λ *q'*. ∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ *list-all2* (|∈|) *qs ps* ∧
(*q* = *q'* ∨ (*q*,*q'*) |∈| (*eps* $\mathcal{A}$)|⁺|))

lemma *ps-reachable-statespE*:

assumes *ps-reachable-statesp* $\mathcal{A}$ *f qs q*
obtains *ps p* **where** *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ *list-all2* (|∈|) *ps qs* ($p = q \vee (p, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$|)
⟨*proof*⟩

lemma *ps-reachable-statesp-Q*:
ps-reachable-statesp $\mathcal{A}$ *f ps q* ⟹ *q* |∈| $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *finite-Collect-ps-statep* [*simp*]:
finite (*Collect* (*ps-reachable-statesp* $\mathcal{A}$ *f ps*)) (**is** *finite ?S*)
⟨*proof*⟩
lemmas *finite-Collect-ps-statep-unfolded* [*simp*] = *finite-Collect-ps-statep*[*unfolded ps-reachable-statesp-def*, *simplified*]

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

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

lemma *ps-reachable-states-fmember*:
q' |∈| *ps-reachable-states* $\mathcal{A}$ *f ps* ⟷ (∃ *qs q*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧
list-all2 (|∈|) *qs ps* ∧ ($q = q' \vee (q, q')$ |∈| (*eps* $\mathcal{A}$)|$^+$|))
⟨*proof*⟩

lemma *ps-reachable-statesI*:
assumes *TA-rule f ps p* |∈| *rules* $\mathcal{A}$ *list-all2* (|∈|) *ps qs* ($p = q \vee (p, q)$ |∈| (*eps* $\mathcal{A}$)|$^+$|)
shows *p* |∈| *ps-reachable-states* $\mathcal{A}$ *f qs*
⟨*proof*⟩

lemma *ps-reachable-states-sig*:
ps-reachable-states $\mathcal{A}$ *f ps* ≠ {||} ⟹ (*f*, *length ps*) |∈| *ta-sig* $\mathcal{A}$
⟨*proof*⟩

A set of "powerset states" is complete if it is sufficient to capture all (non)deterministic derivations.

definition *ps-states-complete-it* :: ('*a*, '*b*) *ta* ⇒ '*a FSet-Lex-Wrapper fset* ⇒ '*a FSet-Lex-Wrapper fset* ⇒ *bool*
where *ps-states-complete-it* $\mathcal{A}$ *Q Qnext* ≡
∀*f ps*. *fset-of-list ps* |⊆| *Q* ∧ *ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*) ≠ {||} ⟶ *Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*)) |∈| *Qnext*

lemma *ps-states-complete-itD*:
ps-states-complete-it $\mathcal{A}$ *Q Qnext* ⟹ *fset-of-list ps* |⊆| *Q* ⟹
ps-reachable-states $\mathcal{A}$ *f* (*map ex ps*) ≠ {||} ⟹ *Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex ps*)) |∈| *Qnext*
⟨*proof*⟩

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

The least complete set of states

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

lemma *ps-states-Pow*:
$ps\text{-}states\text{-}set\ \mathcal{A} \subseteq fset\ (Wrapp\ |`|\ fPow\ (\mathcal{Q}\ \mathcal{A}))$
⟨*proof*⟩

context
includes *fset.lifting*
begin
lift-definition *ps-states* :: ('a, 'b) *ta* ⇒ 'a *FSet-Lex-Wrapper fset* **is** *ps-states-set*
⟨*proof*⟩

lemma *ps-states*: $ps\text{-}states\ \mathcal{A}\ |\subseteq|\ Wrapp\ |`|\ fPow\ (\mathcal{Q}\ \mathcal{A})$ ⟨*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*:
$ps\text{-}states\text{-}complete\ \mathcal{A}\ (ps\text{-}states\ \mathcal{A})$
⟨*proof*⟩

lemma *ps-states-least-complete*:
assumes $ps\text{-}states\text{-}complete\text{-}it\ \mathcal{A}\ Q\ Qnext\ Qnext\ |\subseteq|\ Q$
shows $ps\text{-}states\ \mathcal{A}\ |\subseteq|\ Q$
⟨*proof*⟩

definition *ps-rulesp* :: ('a, 'b) *ta* ⇒ 'a *FSet-Lex-Wrapper fset* ⇒ ('a *FSet-Lex-Wrapper*, 'b) *ta-rule* ⇒ *bool* **where**
$ps\text{-}rulesp\ \mathcal{A}\ Q = (\lambda\ r.\ \exists\ f\ ps\ p.\ r = TA\text{-}rule\ f\ ps\ (Wrapp\ p) \land fset\text{-}of\text{-}list\ ps\ |\subseteq|$
$Q \land$
$p = ps\text{-}reachable\text{-}states\ \mathcal{A}\ f\ (map\ ex\ ps) \land p \neq \{||\})$

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

lemma *finite-ps-rulesp* [*simp*]:
finite (*Collect* (*ps-rulesp* $\mathcal{A}$ *Q*)) (**is** *finite ?S*)
⟨*proof*⟩

lemmas *finite-ps-rulesp-unfolded* = *finite-ps-rulesp*[*unfolded ps-rulesp-def*, *simplified*]

lemmas *ps-rulesI* [*intro*!] = *fCollect-memberI*[*OF finite-ps-rulesp*]

lemma *ps-rules-states*:
rule-states (*fCollect* (*ps-rulesp* $\mathcal{A}$ *Q*)) |⊆| (*Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$) |∪| *Q*)
⟨*proof*⟩

definition *ps-ta* :: (′*q*, ′*f*) *ta* ⇒ (′*q FSet-Lex-Wrapper*, ′*f*) *ta* **where**
ps-ta $\mathcal{A}$ = (*let Q* = *ps-states* $\mathcal{A}$ *in*
TA (*ps-rules* $\mathcal{A}$ *Q*) {||})

definition *ps-ta-*Q_f :: ′*q fset* ⇒ (′*q*, ′*f*) *ta* ⇒ ′*q FSet-Lex-Wrapper fset* **where**
ps-ta-Q_f *Q* $\mathcal{A}$ = (*let Q*′ = *ps-states* $\mathcal{A}$ *in*
ffilter (λ *S*. *Q* |∩| (*ex S*) ≠ {||}) *Q*′)

lemma *ps-rules-sound*:
assumes *p* |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*)
shows *ex p* |⊆| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟨*proof*⟩

lemma *ps-ta-nt-empty-set*:
TA-rule f qs (*Wrapp* {||}) |∈| *rules* (*ps-ta* $\mathcal{A}$) ⟹ *False*
⟨*proof*⟩

lemma *ps-rules-not-empty-reach*:
assumes *Wrapp* {||} |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*)
shows *False* ⟨*proof*⟩

lemma *ps-rules-complete*:
assumes *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
shows ∃ *p*. *q* |∈| *ex p* ∧ *p* |∈| *ta-der* (*ps-ta* $\mathcal{A}$) (*term-of-gterm t*) ∧ *p* |∈| *ps-states* $\mathcal{A}$ ⟨*proof*⟩

lemma *ps-ta-eps*[*simp*]: *eps* (*ps-ta* $\mathcal{A}$) = {||} ⟨*proof*⟩

lemma *ps-ta-det*[*iff*]: *ta-det* (*ps-ta* $\mathcal{A}$) ⟨*proof*⟩

lemma *ps-gta-lang*:
gta-lang (*ps-ta-*Q_f *Q* $\mathcal{A}$) (*ps-ta* $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$ (**is** *?R* = *?L*)
⟨*proof*⟩

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

lemma $\mathcal{L}$*-ps-reg*:
$\mathcal{L}$ (*ps-reg R*) = $\mathcal{L}$ *R*
⟨*proof*⟩

lemma *ps-ta-states*: $\mathcal{Q}$ (*ps-ta* $\mathcal{A}$) |⊆| *Wrapp* |‘| *fPow* ($\mathcal{Q}$ $\mathcal{A}$)
⟨*proof*⟩

lemma *ps-ta-states′*: *ex* |'| $\mathcal{Q}$ (*ps-ta* $\mathcal{A}$) |⊆| *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-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ ⟷
(∀ *t*. *funas-gterm t* ⊆ *fset* $\mathcal{F}$ ⟷ (∃ *q*. *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*)))

definition *sig-ta* **where**
sig-ta $\mathcal{F}$ = *TA* ((λ (*f*, *n*). *TA-rule f* (*replicate n* ()) ()) |'| $\mathcal{F}$) {||}

lemma *sig-ta-rules-fmember*:
TA-rule f qs q |∈| *rules* (*sig-ta* $\mathcal{F}$) ⟷ (∃ *n*. (*f*, *n*) |∈| $\mathcal{F}$ ∧ *qs* = *replicate n* ()
∧ *q* = ())
⟨*proof*⟩

lemma *sig-ta-completely-defined*:
partially-completely-defined-on (*sig-ta* $\mathcal{F}$) $\mathcal{F}$
⟨*proof*⟩

lemma *ta-der-fsubset-sig-ta-completely*:
assumes *ta-subset* (*sig-ta* $\mathcal{F}$) $\mathcal{A}$ *ta-sig* $\mathcal{A}$ |⊆| $\mathcal{F}$
shows *partially-completely-defined-on* $\mathcal{A}$ $\mathcal{F}$
⟨*proof*⟩

lemma *completely-definied-ps-taI*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ ⟹ *partially-completely-defined-on* (*ps-ta* $\mathcal{A}$)
$\mathcal{F}$
⟨*proof*⟩

lemma *completely-definied-ta-union1I*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ ⟹ *ta-sig* $\mathcal{B}$ |⊆| $\mathcal{F}$ ⟹ $\mathcal{Q}$ $\mathcal{A}$ |∩| $\mathcal{Q}$ $\mathcal{B}$ = {||}
⟹
partially-completely-defined-on (*ta-union* $\mathcal{A}$ $\mathcal{B}$) $\mathcal{F}$
⟨*proof*⟩

lemma *completely-definied-fmaps-statesI*:
partially-completely-defined-on $\mathcal{A}$ $\mathcal{F}$ ⟹ *finj-on f* ($\mathcal{Q}$ $\mathcal{A}$) ⟹ *partially-completely-defined-on*
(*fmap-states-ta f* $\mathcal{A}$) $\mathcal{F}$
⟨*proof*⟩

lemma *det-completely-defined-complement*:
assumes *partially-completely-defined-on* $\mathcal{A}$ $\mathcal{F}$ *ta-det* $\mathcal{A}$

shows *gta-lang* ($\mathcal{Q}$ $\mathcal{A}$ |−| Q) $\mathcal{A}$ = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − *gta-lang* Q $\mathcal{A}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *ta-der-gterm-sig-fset*:
q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ⟹ *funas-gterm t* ⊆ *fset* (*ta-sig* $\mathcal{A}$)
⟨*proof*⟩

definition *filter-ta-sig* **where**
filter-ta-sig $\mathcal{F}$ $\mathcal{A}$ = *TA* (*ffilter* (λ *r*. (*r-root r*, *length* (*r-lhs-states r*)) |∈| $\mathcal{F}$) (*rules* $\mathcal{A}$)) (*eps* $\mathcal{A}$)

definition *filter-ta-reg* **where**
filter-ta-reg $\mathcal{F}$ *R* = *Reg* (*fin R*) (*filter-ta-sig* $\mathcal{F}$ (*ta R*))

lemma *filter-ta-sig*:
ta-sig (*filter-ta-sig* $\mathcal{F}$ $\mathcal{A}$) |⊆| $\mathcal{F}$
⟨*proof*⟩

lemma *filter-ta-sig-lang*:
gta-lang Q (*filter-ta-sig* $\mathcal{F}$ $\mathcal{A}$) = *gta-lang Q* $\mathcal{A}$ ∩ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma $\mathcal{L}$*-filter-ta-reg*:
$\mathcal{L}$ (*filter-ta-reg* $\mathcal{F}$ $\mathcal{A}$) = $\mathcal{L}$ $\mathcal{A}$ ∩ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
⟨*proof*⟩

definition *sig-ta-reg* **where**
sig-ta-reg $\mathcal{F}$ = *Reg* {||} (*sig-ta* $\mathcal{F}$)

lemma $\mathcal{L}$*-sig-ta-reg*:
$\mathcal{L}$ (*sig-ta-reg* $\mathcal{F}$) = {}
⟨*proof*⟩

definition *complement-reg* **where**
complement-reg R $\mathcal{F}$ = (*let* $\mathcal{A}$ = *ps-reg* (*reg-union* (*sig-ta-reg* $\mathcal{F}$) *R*) *in*
Reg ($\mathcal{Q}_r$ $\mathcal{A}$ |−| *fin* $\mathcal{A}$) (*ta* $\mathcal{A}$))

lemma $\mathcal{L}$*-complement-reg*:
assumes *ta-sig* (*ta* $\mathcal{A}$) |⊆| $\mathcal{F}$
shows $\mathcal{L}$ (*complement-reg* $\mathcal{A}$ $\mathcal{F}$) = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

lemma $\mathcal{L}$*-complement-filter-reg*:
$\mathcal{L}$ (*complement-reg* (*filter-ta-reg* $\mathcal{F}$ $\mathcal{A}$) $\mathcal{F}$) = $\mathcal{T}_G$ (*fset* $\mathcal{F}$) − $\mathcal{L}$ $\mathcal{A}$
⟨*proof*⟩

definition *difference-reg* **where**
difference-reg R L = (*let F* = *ta-sig* (*ta R*) *in*
reg-intersect R (*trim-reg* (*complement-reg* (*filter-ta-reg F L*) *F*)))

lemma $\mathcal{L}$-*difference-reg*:
$\mathcal{L}$ (*difference-reg R L*) = $\mathcal{L}$ *R* − $\mathcal{L}$ *L* (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

end
theory *Tree-Automata-Pumping*
imports *Tree-Automata*
begin

3.4 Pumping lemma

abbreviation *derivation-ctxt ts Cs* ≡ *Suc* (*length Cs*) = *length ts* ∧
(∀ *i* < *length Cs*. (*Cs* ! *i*) ⟨*ts* ! *i*⟩ = *ts* ! *Suc i*)

abbreviation *derivation-ctxt-st A ts Cs qs* ≡ *length qs* = *length ts* ∧ *Suc* (*length Cs*) = *length ts* ∧
(∀ *i* < *length Cs*. *qs* ! *Suc i* |∈| *ta-der A* (*Cs* ! *i*)⟨*Var* (*qs* ! *i*)⟩)

abbreviation *derivation-sound A ts qs* ≡ *length qs* = *length ts* ∧
(∀ *i* < *length qs*. *qs* ! *i* |∈| *ta-der A* (*ts* ! *i*))

definition *derivation A ts Cs qs* ⟷ *derivation-ctxt ts Cs* ∧
derivation-ctxt-st A ts Cs qs ∧ *derivation-sound A ts qs*

lemma *ctxt-comp-lhs-not-hole*:
assumes *C* ≠ □
shows *C* $\circ_c$ *D* ≠ □
⟨*proof*⟩

lemma *ctxt-comp-rhs-not-hole*:
assumes *D* ≠ □
shows *C* $\circ_c$ *D* ≠ □
⟨*proof*⟩

lemma *fold-ctxt-comp-nt-empty-acc*:
assumes *D* ≠ □
shows *fold* ($\circ_c$) *Cs D* ≠ □
⟨*proof*⟩

lemma *fold-ctxt-comp-nt-empty*:
assumes *C* ∈ *set Cs* **and** *C* ≠ □
shows *fold* ($\circ_c$) *Cs D* ≠ □ ⟨*proof*⟩

lemma *empty-ctxt-power* [*simp*]:

$\Box \widehat{\ } n = \Box$
⟨*proof*⟩

lemma *ctxt-comp-not-hole*:
assumes $C \neq \Box$ **and** $n \neq 0$
shows $C \widehat{\ } n \neq \Box$
⟨*proof*⟩

lemma *ctxt-comp-n-suc* [*simp*]:
shows $(C \widehat{\ } (Suc\ n))\langle t\rangle = (C \widehat{\ } n)\langle C\langle t\rangle\rangle$
⟨*proof*⟩

lemma *ctxt-comp-reach*:
assumes $p\ |{\in}|\ ta\text{-}der\ A\ C\langle Var\ p\rangle$
shows $p\ |{\in}|\ ta\text{-}der\ A\ (C \widehat{\ } n)\langle Var\ p\rangle$
⟨*proof*⟩

lemma *args-depth-less* [*simp*]:
assumes $u \in set\ ss$
shows $depth\ u < depth\ (Fun\ f\ ss)$ ⟨*proof*⟩

lemma *subterm-depth-less*:
assumes $s \rhd t$
shows $depth\ t < depth\ s$
⟨*proof*⟩

lemma *poss-length-depth*:
shows $\exists\ p \in poss\ t.\ length\ p = depth\ t$
⟨*proof*⟩

lemma *poss-length-bounded-by-depth*:
assumes $p \in poss\ t$
shows $length\ p \leq depth\ t$ ⟨*proof*⟩

lemma *depth-ctxt-nt-hole-inc*:
assumes $C \neq \Box$
shows $depth\ t < depth\ C\langle t\rangle$ ⟨*proof*⟩

lemma *depth-ctxt-less-eq*:
$depth\ t \leq depth\ C\langle t\rangle$ ⟨*proof*⟩

lemma *ctxt-comp-n-not-hole-depth-inc*:
assumes $C \neq \Box$
shows $depth\ (C \widehat{\ } n)\langle t\rangle < depth\ (C \widehat{\ } (Suc\ n))\langle t\rangle$

⟨*proof*⟩

lemma *ctxt-comp-n-lower-bound*:
assumes $C \neq \Box$
shows $n <$ *depth* $(C\widehat{}(Suc\ n))\langle t\rangle$
⟨*proof*⟩

lemma *ta-der-ctxt-n-loop*:
assumes q |∈| *ta-der* $\mathcal{A}$ t q |∈| *ta-der* $\mathcal{A}$ $C\langle Var\ q\rangle$
shows q |∈| *ta-der* $\mathcal{A}$ $(C\widehat{}n)\langle t\rangle$
⟨*proof*⟩

lemma *ctxt-compose-funs-ctxt* [*simp*]:
funs-ctxt $(C \circ_c D)$ = *funs-ctxt* $C \cup$ *funs-ctxt* D
⟨*proof*⟩

lemma *ctxt-compose-vars-ctxt* [*simp*]:
vars-ctxt $(C \circ_c D)$ = *vars-ctxt* $C \cup$ *vars-ctxt* D
⟨*proof*⟩

lemma *ctxt-power-funs-vars-0* [*simp*]:
assumes $n = 0$
shows *funs-ctxt* $(C\widehat{}n) = \{\}$ *vars-ctxt* $(C\widehat{}n) = \{\}$
⟨*proof*⟩

lemma *ctxt-power-funs-vars-n* [*simp*]:
assumes $n \neq 0$
shows *funs-ctxt* $(C\widehat{}n)$ = *funs-ctxt* C *vars-ctxt* $(C\widehat{}n)$ = *vars-ctxt* C
⟨*proof*⟩

fun *terms-pos* **where**
terms-pos s [] = [s]
| *terms-pos* s (p # ps) = *terms-pos* (s |- [p]) ps @ [s]

lemma *subt-at-poss* [*simp*]:
assumes a # $p \in$ *poss* s
shows $p \in$ *poss* (s |- [a])
⟨*proof*⟩

lemma *terms-pos-length* [*simp*]:
shows *length* (*terms-pos* t p) = *Suc* (*length* p)
⟨*proof*⟩

lemma *terms-pos-last* [*simp*]:
assumes i = *length* p
shows *terms-pos* t p ! i = t ⟨*proof*⟩

lemma *terms-pos-subterm*:
assumes $p \in poss\ t$ **and** $s \in set\ (terms\text{-}pos\ t\ p)$
shows $t \trianglerighteq s$ ⟨*proof*⟩

lemma *terms-pos-differ-subterm*:
assumes $p \in poss\ t$ **and** $i < length\ (terms\text{-}pos\ t\ p)$
and $j < length\ (terms\text{-}pos\ t\ p)$ **and** $i < j$
shows $terms\text{-}pos\ t\ p\ !\ i \vartriangleleft terms\text{-}pos\ t\ p\ !\ j$
⟨*proof*⟩

lemma *distinct-terms-pos*:
assumes $p \in poss\ t$
shows $distinct\ (terms\text{-}pos\ t\ p)$ ⟨*proof*⟩

lemma *term-chain-depth*:
assumes $depth\ t = n$
shows $\exists\ p \in poss\ t.\ length\ (terms\text{-}pos\ t\ p) = (n + 1)$
⟨*proof*⟩

lemma *ta-der-derivation-chain-terms-pos-exist*:
assumes $p \in poss\ t$ **and** $q\ |{\in}|\ ta\text{-}der\ A\ t$
shows $\exists\ Cs\ qs.\ derivation\ A\ (terms\text{-}pos\ t\ p)\ Cs\ qs \wedge last\ qs = q$
⟨*proof*⟩

lemma *derivation-ctxt-terms-pos-nt-empty*:
assumes $p \in poss\ t$ **and** $derivation\text{-}ctxt\ (terms\text{-}pos\ t\ p)\ Cs$ **and** $C \in set\ Cs$
shows $C \neq \Box$
⟨*proof*⟩

lemma *derivation-ctxt-terms-pos-sub-list-nt-empty*:
assumes $p \in poss\ t$ **and** $derivation\text{-}ctxt\ (terms\text{-}pos\ t\ p)\ Cs$
and $i < length\ Cs$ **and** $j \leq length\ Cs$ **and** $i < j$
shows $fold\ (\circ_c)\ (take\ (j - i)\ (drop\ i\ Cs))\ \Box \neq \Box$
⟨*proof*⟩

lemma *derivation-ctxt-comp-term*:
assumes $derivation\text{-}ctxt\ ts\ Cs$
and $i < length\ Cs$ **and** $j \leq length\ Cs$ **and** $i < j$
shows $(fold\ (\circ_c)\ (take\ (j - i)\ (drop\ i\ Cs))\ \Box)\langle ts\ !\ i\rangle = ts\ !\ j$
⟨*proof*⟩

lemma *derivation-ctxt-comp-states*:
assumes $derivation\text{-}ctxt\text{-}st\ A\ ts\ Cs\ qs$
and $i < length\ Cs$ **and** $j \leq length\ Cs$ **and** $i < j$
shows $qs\ !\ j\ |{\in}|\ ta\text{-}der\ A\ (fold\ (\circ_c)\ (take\ (j - i)\ (drop\ i\ Cs))\ \Box)\langle Var\ (qs\ !\ i)\rangle$
⟨*proof*⟩

lemma *terms-pos-ground*:
assumes *ground t* **and** $p \in$ *poss t*
shows $\forall$ $s \in$ *set* (*terms-pos t p*). *ground s*
⟨*proof*⟩

lemma *list-card-smaller-contains-eq-elemens*:
assumes *length qs* = *n* **and** *card* (*set qs*) < *n*
shows $\exists$ *i* < *length qs*. $\exists$ *j* < *length qs*. *i* < *j* $\wedge$ *qs* ! *i* = *qs* ! *j*
⟨*proof*⟩

lemma *length-remdups-less-eq*:
assumes *set xs* $\subseteq$ *set ys*
shows *length* (*remdups xs*) $\leq$ *length* (*remdups ys*) ⟨*proof*⟩

lemma *pigeonhole-tree-automata*:
assumes *fcard* ($\mathcal{Q}$ *A*) < *depth t* **and** *q* |∈| *ta-der A t* **and** *ground t*
shows $\exists$ *C C2 v p*. *C2* $\neq$ □ $\wedge$ *C*⟨*C2*⟨*v*⟩⟩ = *t* $\wedge$ *p* |∈| *ta-der A v* $\wedge$
p |∈| *ta-der A C2*⟨*Var p*⟩ $\wedge$ *q* |∈| *ta-der A C*⟨*Var p*⟩
⟨*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 *ta-der* $\mathcal{A}$ (*term-of-gterm s*) = *ta-der* $\mathcal{A}$ (*term-of-gterm t*)
shows *ta-der* $\mathcal{A}$ (*term-of-gterm* $C\langle s\rangle_G$) = *ta-der* $\mathcal{A}$ (*term-of-gterm* $C\langle t\rangle_G$) ⟨*proof*⟩

locale *myhill-nerode* =
fixes $\mathcal{F}$ $\mathcal{L}$ **assumes** *term-subset*: $\mathcal{L} \subseteq \mathcal{T}_G\ \mathcal{F}$
begin

definition *myhill* (- $\equiv_{\mathcal{L}}$ -) **where**
myhill s t $\equiv s \in \mathcal{T}_G\ \mathcal{F} \wedge t \in \mathcal{T}_G\ \mathcal{F} \wedge (\forall\ C.\ C\langle s\rangle_G \in \mathcal{L} \wedge C\langle t\rangle_G \in \mathcal{L} \vee C\langle s\rangle_G \notin \mathcal{L} \wedge C\langle t\rangle_G \notin \mathcal{L})$

lemma *myhill-sound*: $s \equiv_{\mathcal{L}} t \Longrightarrow s \in \mathcal{T}_G\ \mathcal{F}$ $\ s \equiv_{\mathcal{L}} t \Longrightarrow t \in \mathcal{T}_G\ \mathcal{F}$
⟨*proof*⟩

lemma *myhill-refl* [*simp*]: $s \in \mathcal{T}_G\ \mathcal{F} \Longrightarrow s \equiv_{\mathcal{L}} s$
⟨*proof*⟩

lemma *myhill-symmetric*: $s \equiv_{\mathcal{L}} t \Longrightarrow t \equiv_{\mathcal{L}} s$

$\langle proof \rangle$

lemma *myhill-trans* [*trans*]:
$s \equiv_{\mathcal{L}} t \Longrightarrow t \equiv_{\mathcal{L}} u \Longrightarrow s \equiv_{\mathcal{L}} u$
$\langle proof \rangle$

abbreviation *myhill-r* ($MN_{\mathcal{L}}$) **where**
myhill-r $\equiv \{(s, t) \mid s\ t.\ s \equiv_{\mathcal{L}} t\}$

lemma *myhill-equiv*:
equiv $(\mathcal{T}_G\ \mathcal{F})\ MN_{\mathcal{L}}$
$\langle proof \rangle$

lemma *rtl-der-image-on-myhill-inj*:
assumes *gta-lang* $Q_f\ \mathcal{A} = \mathcal{L}$
shows *inj-on* (λ X. *gta-der* $\mathcal{A}$ ‘ X) $(\mathcal{T}_G\ \mathcal{F}\ //\ MN_{\mathcal{L}})$ (**is** *inj-on ?D ?R*)
$\langle proof \rangle$

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*)
$\langle proof \rangle$

end

end
theory *GTT*
imports *Tree-Automata Ground-Closure*
begin

4 Ground Tree Transducers (GTT)

type-synonym (*'q*, *'f*) *gtt* = (*'q*, *'f*) *ta* × (*'q*, *'f*) *ta*

abbreviation *gtt-rules* **where**
gtt-rules $\mathcal{G} \equiv$ *rules* (*fst* $\mathcal{G}$) |∪| *rules* (*snd* $\mathcal{G}$)
abbreviation *gtt-eps* **where**
gtt-eps $\mathcal{G} \equiv$ *eps* (*fst* $\mathcal{G}$) |∪| *eps* (*snd* $\mathcal{G}$)
definition *gtt-states* **where**
gtt-states $\mathcal{G} = \mathcal{Q}$ (*fst* $\mathcal{G}$) |∪| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
abbreviation *gtt-syms* **where**
gtt-syms $\mathcal{G} \equiv$ *ta-sig* (*fst* $\mathcal{G}$) |∪| *ta-sig* (*snd* $\mathcal{G}$)
definition *gtt-interface* **where**
gtt-interface $\mathcal{G} = \mathcal{Q}$ (*fst* $\mathcal{G}$) |∩| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
definition *gtt-eps-free* **where**
gtt-eps-free $\mathcal{G}$ = (*eps-free* (*fst* $\mathcal{G}$), *eps-free* (*snd* $\mathcal{G}$))

definition *is-gtt-eps-free* :: (*'q*, *'f*) *ta* × (*'p*, *'g*) *ta* ⇒ *bool* **where**
is-gtt-eps-free $\mathcal{G} \longleftrightarrow$ *eps* (*fst* $\mathcal{G}$) = {||} ∧ *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) |t u q. q |∈| gta-der (fst $\mathcal{G}$) t ∧ q |∈| gta-der (snd $\mathcal{G}$) u}

lemma *agtt-langI*:
q |∈| gta-der (fst $\mathcal{G}$) s ⟹ q |∈| gta-der (snd $\mathcal{G}$) t ⟹ (s, t) ∈ 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 $\mathcal{G}$ ≡ gmctxt-cl UNIV (agtt-lang $\mathcal{G}$)

lemma *gtt-lang-join*:
q |∈| gta-der (fst $\mathcal{G}$) s ⟹ q |∈| gta-der (snd $\mathcal{G}$) t ⟹ (s, t) ∈ gmctxt-cl UNIV (agtt-lang $\mathcal{G}$)
⟨proof⟩

definition *gtt-accept* **where**
gtt-accept $\mathcal{G}$ s t ≡ (s, t) ∈ gmctxt-cl UNIV (agtt-lang $\mathcal{G}$)

lemma *gtt-accept-intros*:
(s, t) ∈ agtt-lang $\mathcal{G}$ ⟹ gtt-accept $\mathcal{G}$ s t
length ss = length ts ⟹ ∀ i < length ts. gtt-accept $\mathcal{G}$ (ss ! i) (ts ! i) ⟹
(f, length ss) ∈ $\mathcal{F}$ ⟹ gtt-accept $\mathcal{G}$ (GFun f ss) (GFun f ts)
⟨proof⟩

abbreviation *gtt-lang-terms* :: ('q, 'f) gtt ⇒ ('f, 'q) term rel **where**
gtt-lang-terms $\mathcal{G}$ ≡ (λ s. map-both term-of-gterm s) ‘ (gmctxt-cl UNIV (agtt-lang $\mathcal{G}$))

lemma *term-of-gterm-gtt-lang-gtt-lang-terms-conv*:
map-both term-of-gterm ‘ gtt-lang $\mathcal{G}$ = gtt-lang-terms $\mathcal{G}$
⟨proof⟩

lemma *gtt-accept-swap* [*simp*]:
gtt-accept (prod.swap $\mathcal{G}$) s t ⟷ gtt-accept $\mathcal{G}$ t s

⟨*proof*⟩

lemma *gtt-lang-swap*:
(*gtt-lang* (*A*, *B*))$^{-1}$ = *gtt-lang* (*B*, *A*)
⟨*proof*⟩

lemma *gtt-accept-exI*:
assumes *gtt-accept* $\mathcal{G}$ *s t*
shows ∃ *u*. *u* |∈| *ta-der′* (*fst* $\mathcal{G}$) (*term-of-gterm s*) ∧ *u* |∈| *ta-der′* (*snd* $\mathcal{G}$) (*term-of-gterm t*)
⟨*proof*⟩

lemma *agtt-lang-mono*:
assumes *rules* (*fst* $\mathcal{G}$) |⊆| *rules* (*fst* $\mathcal{G}'$) *eps* (*fst* $\mathcal{G}$) |⊆| *eps* (*fst* $\mathcal{G}'$)
rules (*snd* $\mathcal{G}$) |⊆| *rules* (*snd* $\mathcal{G}'$) *eps* (*snd* $\mathcal{G}$) |⊆| *eps* (*snd* $\mathcal{G}'$)
shows *agtt-lang* $\mathcal{G}$ ⊆ *agtt-lang* $\mathcal{G}'$
⟨*proof*⟩

lemma *gtt-lang-mono*:
assumes *rules* (*fst* $\mathcal{G}$) |⊆| *rules* (*fst* $\mathcal{G}'$) *eps* (*fst* $\mathcal{G}$) |⊆| *eps* (*fst* $\mathcal{G}'$)
rules (*snd* $\mathcal{G}$) |⊆| *rules* (*snd* $\mathcal{G}'$) *eps* (*snd* $\mathcal{G}$) |⊆| *eps* (*snd* $\mathcal{G}'$)
shows *gtt-lang* $\mathcal{G}$ ⊆ *gtt-lang* $\mathcal{G}'$
⟨*proof*⟩

definition *fmap-states-gtt* **where**
fmap-states-gtt f ≡ *map-both* (*fmap-states-ta f*)

lemma *ground-map-vars-term-simp*:
ground t ⟹ *map-term f g t* = *map-term f* (λ-. *undefined*) *t*
⟨*proof*⟩

lemma *states-fmap-states-gtt* [*simp*]:
gtt-states (*fmap-states-gtt f* $\mathcal{G}$) = *f* |‘| *gtt-states* $\mathcal{G}$
⟨*proof*⟩

lemma *agtt-lang-fmap-states-gtt*:
assumes *finj-on f* (*gtt-states* $\mathcal{G}$)
shows *agtt-lang* (*fmap-states-gtt f* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *agtt-lang-Inl-Inr-states-agtt*:
agtt-lang (*fmap-states-gtt Inl* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
agtt-lang (*fmap-states-gtt Inr* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *gtt-lang-fmap-states-gtt*:

assumes *finj-on f* (*gtt-states* $\mathcal{G}$)
shows *gtt-lang* (*fmap-states-gtt f* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

definition *gtt-only-reach* **where**
gtt-only-reach = *map-both ta-only-reach*

4.1 (A)GTT reachable states

lemma *agtt-only-reach-lang*:
agtt-lang (*gtt-only-reach* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *gtt-only-reach-lang*:
gtt-lang (*gtt-only-reach* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *gtt-only-reach-syms*:
gtt-syms (*gtt-only-reach* $\mathcal{G}$) |⊆| *gtt-syms* $\mathcal{G}$
⟨*proof*⟩

4.2 (A)GTT productive states

definition *gtt-only-prod* **where**
gtt-only-prod $\mathcal{G}$ = (*let iface* = *gtt-interface* $\mathcal{G}$ *in*
map-both (*ta-only-prod iface*) $\mathcal{G}$)

lemma *agtt-only-prod-lang*:
agtt-lang (*gtt-only-prod* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *gtt-only-prod-lang*:
gtt-lang (*gtt-only-prod* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *gtt-only-prod-syms*:
gtt-syms (*gtt-only-prod* $\mathcal{G}$) |⊆| *gtt-syms* $\mathcal{G}$
⟨*proof*⟩

4.3 (A)GTT trimming

definition *trim-gtt* **where**
trim-gtt = *gtt-only-prod* ∘ *gtt-only-reach*

lemma *trim-agtt-lang*:
agtt-lang (*trim-gtt G*) = *agtt-lang G*
⟨*proof*⟩

lemma *trim-gtt-lang*:
gtt-lang (*trim-gtt G*) = *gtt-lang G*

⟨*proof*⟩

lemma *trim-gtt-prod-syms*:
gtt-syms (*trim-gtt* *G*) |⊆| *gtt-syms* *G*
⟨*proof*⟩

4.4 root-cleanliness

A GTT is root-clean if none of its interface states can occur in a non-root positions in the accepting derivations corresponding to its anchored GTT relation.

definition *ta-nr-states* :: ('*q*, '*f*) *ta* ⇒ '*q* *fset* **where**
ta-nr-states *A* = |⋃| ((*fset-of-list* ∘ *r-lhs-states*) |'| (*rules* *A*))

definition *gtt-nr-states* **where**
gtt-nr-states *G* = *ta-nr-states* (*fst* *G*) |∪| *ta-nr-states* (*snd* *G*)

definition *gtt-root-clean* **where**
gtt-root-clean *G* ⟷ *gtt-nr-states* *G* |∩| *gtt-interface* *G* = {||}

4.5 Relabeling

definition *relabel-gtt* :: ('*q* :: *linorder*, '*f*) *gtt* ⇒ (*nat*, '*f*) *gtt* **where**
relabel-gtt *G* = *fmap-states-gtt* (*map-fset-to-nat* (*gtt-states* *G*)) *G*

lemma *relabel-agtt-lang* [*simp*]:
agtt-lang (*relabel-gtt* *G*) = *agtt-lang* *G*
⟨*proof*⟩

lemma *agtt-lang-sig*:
fset (*gtt-syms* *G*) ⊆ $\mathcal{F}$ ⟹ *agtt-lang* *G* ⊆ $\mathcal{T}_G$ $\mathcal{F}$ × $\mathcal{T}_G$ $\mathcal{F}$
⟨*proof*⟩

4.6 epsilon free GTTs

lemma *agtt-lang-gtt-eps-free* [*simp*]:
agtt-lang (*gtt-eps-free* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *gtt-lang-gtt-eps-free* [*simp*]:
gtt-lang (*gtt-eps-free* $\mathcal{G}$) = *gtt-lang* $\mathcal{G}$
⟨*proof*⟩

end
theory *GTT-Compose*
imports *GTT*
begin

4.7 GTT closure under composition

inductive-set Δ_ε*-set* :: $('q, 'f)$ *ta* $\Rightarrow$ $('q, 'f)$ *ta* $\Rightarrow$ $('q \times 'q)$ *set* **for** $\mathcal{A}$ $\mathcal{B}$ **where**
Δ_ε*-set-cong*: *TA-rule f ps p* $|\in|$ *rules* $\mathcal{A} \Longrightarrow$ *TA-rule f qs q* $|\in|$ *rules* $\mathcal{B} \Longrightarrow$ *length ps* = *length qs* $\Longrightarrow$
$(\bigwedge i.\ i <$ *length qs* $\Longrightarrow$ (*ps* ! *i*, *qs* ! *i*) $\in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B}$) $\Longrightarrow$ $(p, q) \in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B}$
| Δ_ε*-set-eps1*: (p, p') $|\in|$ *eps* $\mathcal{A} \Longrightarrow (p, q) \in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B} \Longrightarrow (p', q) \in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B}$
| Δ_ε*-set-eps2*: (q, q') $|\in|$ *eps* $\mathcal{B} \Longrightarrow (p, q) \in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B} \Longrightarrow (p, q') \in \Delta_\varepsilon$*-set* $\mathcal{A}$ $\mathcal{B}$

lemma Δ_ε*-states*: Δ_ε*-set* $\mathcal{A}$ $\mathcal{B} \subseteq$ *fset* ($\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathcal{B}$)
⟨*proof*⟩

lemma *finite-*Δ_ε [*simp*]: *finite* (Δ_ε*-set* $\mathcal{A}$ $\mathcal{B}$)
⟨*proof*⟩

context
includes *fset.lifting*
begin
lift-definition Δ_ε :: $('q, 'f)$ *ta* $\Rightarrow$ $('q, 'f)$ *ta* $\Rightarrow$ $('q \times 'q)$ *fset* **is** Δ_ε*-set* ⟨*proof*⟩
lemmas Δ_ε*-cong* = Δ_ε*-set-cong* [*Transfer.transferred*]
lemmas Δ_ε*-eps1* = Δ_ε*-set-eps1* [*Transfer.transferred*]
lemmas Δ_ε*-eps2* = Δ_ε*-set-eps2* [*Transfer.transferred*]
lemmas Δ_ε*-cases* = Δ_ε*-set.cases*[*Transfer.transferred*]
lemmas Δ_ε*-induct* [*consumes 1*, *case-names* Δ_ε*-cong* Δ_ε*-eps1* Δ_ε*-eps2*] = Δ_ε*-set.induct*[*Transfer.transferred*]
lemmas Δ_ε*-intros* = Δ_ε*-set.intros*[*Transfer.transferred*]
lemmas Δ_ε*-simps* = Δ_ε*-set.simps*[*Transfer.transferred*]
end

lemma *finite-alt-def* [*simp*]:
finite $\{(\alpha, \beta).\ (\exists t.$ *ground t* $\wedge\ \alpha$ $|\in|$ *ta-der* $\mathcal{A}$ *t* $\wedge\ \beta$ $|\in|$ *ta-der* $\mathcal{B}$ *t*$)\}$ (**is** *finite ?S*)
⟨*proof*⟩

lemma Δ_ε*-def′*:
Δ_ε $\mathcal{A}$ $\mathcal{B}$ = $\{|(\alpha, \beta).\ (\exists t.$ *ground t* $\wedge\ \alpha$ $|\in|$ *ta-der* $\mathcal{A}$ *t* $\wedge\ \beta$ $|\in|$ *ta-der* $\mathcal{B}$ *t*$)|\}$
⟨*proof*⟩

lemma Δ_ε*-fmember*:
(p, q) $|\in|$ Δ_ε $\mathcal{A}$ $\mathcal{B} \longleftrightarrow (\exists t.$ *ground t* $\wedge$ *p* $|\in|$ *ta-der* $\mathcal{A}$ *t* $\wedge$ *q* $|\in|$ *ta-der* $\mathcal{B}$ *t*$)$
⟨*proof*⟩

definition *GTT-comp* :: $('q, 'f)$ *gtt* $\Rightarrow$ $('q, 'f)$ *gtt* $\Rightarrow$ $('q, 'f)$ *gtt* **where**
GTT-comp $\mathcal{G}_1$ $\mathcal{G}_2$ =
(*let* $\Delta = \Delta_\varepsilon$ (*snd* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$) *in*
(*TA* (*gtt-rules* (*fst* $\mathcal{G}_1$, *fst* $\mathcal{G}_2$)) (*eps* (*fst* $\mathcal{G}_1$) $|\cup|$ *eps* (*fst* $\mathcal{G}_2$) $|\cup|$ Δ),
TA (*gtt-rules* (*snd* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$)) (*eps* (*snd* $\mathcal{G}_1$) $|\cup|$ *eps* (*snd* $\mathcal{G}_2$) $|\cup|$ $(\Delta|^{-1}|)$)))

lemma *gtt-syms-GTT-comp*:
gtt-syms (*GTT-comp A B*) = *gtt-syms A* $|\cup|$ *gtt-syms B*
⟨*proof*⟩

lemma Δ_ε-*statesD*:
(p, q) |∈| Δ_ε $\mathcal{A}$ $\mathcal{B}$ $\Longrightarrow$ p |∈| $\mathcal{Q}$ $\mathcal{A}$
(p, q) |∈| Δ_ε $\mathcal{A}$ $\mathcal{B}$ $\Longrightarrow$ q |∈| $\mathcal{Q}$ $\mathcal{B}$
⟨*proof*⟩

lemma Δ_ε-*statesD'*:
q |∈| *eps-states* (Δ_ε $\mathcal{A}$ $\mathcal{B}$) $\Longrightarrow$ q |∈| $\mathcal{Q}$ $\mathcal{A}$ |∪| $\mathcal{Q}$ $\mathcal{B}$
⟨*proof*⟩

lemma Δ_ε-*swap*:
prod.swap p |∈| Δ_ε $\mathcal{A}$ $\mathcal{B}$ $\longleftrightarrow$ p |∈| Δ_ε $\mathcal{B}$ $\mathcal{A}$
⟨*proof*⟩

lemma Δ_ε-*inverse* [*simp*]:
(Δ_ε $\mathcal{A}$ $\mathcal{B}$)|$^{-1}$| = Δ_ε $\mathcal{B}$ $\mathcal{A}$
⟨*proof*⟩

lemma *gtt-states-comp-union*:
gtt-states (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) |⊆| *gtt-states* $\mathcal{G}_1$ |∪| *gtt-states* $\mathcal{G}_2$
⟨*proof*⟩

lemma *GTT-comp-swap* [*simp*]:
GTT-comp (*prod.swap* $\mathcal{G}_2$) (*prod.swap* $\mathcal{G}_1$) = *prod.swap* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)
⟨*proof*⟩

lemma *gtt-comp-complete-semi*:
assumes *s*: q |∈| *gta-der* (*fst* $\mathcal{G}_1$) s **and** *u*: q |∈| *gta-der* (*snd* $\mathcal{G}_1$) u **and** *ut*: *gtt-accept* $\mathcal{G}_2$ u t
shows q |∈| *gta-der* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) s q |∈| *gta-der* (*snd* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)) t
⟨*proof*⟩

lemmas *gtt-comp-complete-semi'* = *gtt-comp-complete-semi*[*of* - *prod.swap* $\mathcal{G}_2$ - - *prod.swap* $\mathcal{G}_1$ **for** $\mathcal{G}_1$ $\mathcal{G}_2$,
unfolded fst-swap snd-swap GTT-comp-swap gtt-accept-swap]

lemma *gtt-comp-acomplete*:
gcomp-rel UNIV (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$) ⊆ *agtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)
⟨*proof*⟩

lemma Δ_ε-*steps-from*-$\mathcal{G}_2$:
assumes (q, q') |∈| (*eps* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)))|$^+$| q |∈| *gtt-states* $\mathcal{G}_2$
gtt-states $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows (q, q') |∈| (*eps* (*fst* $\mathcal{G}_2$))|$^+$| ∧ q' |∈| *gtt-states* $\mathcal{G}_2$
⟨*proof*⟩

lemma Δ_ε-*steps-from*-$\mathcal{G}_1$:
assumes (p, r) |∈| (*eps* (*fst* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)))|$^+$| p |∈| *gtt-states* $\mathcal{G}_1$

$gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
obtains $r\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ (p,\ r)\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$|\ q\ p'$ **where** $r\ |\in|\ gtt\text{-}states\ \mathcal{G}_2\ p = p' \vee (p,\ p')\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|\ (p',\ q)\ |\in|\ \Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2)$
$q = r \vee (q,\ r)\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+|$
$\langle proof \rangle$

lemma $\Delta_\varepsilon\text{-}steps\text{-}from\text{-}\mathcal{G}_1\text{-}\mathcal{G}_2$:
assumes $(q,\ q')\ |\in|\ (eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+|\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|\ gtt\text{-}states\ \mathcal{G}_2$
$gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
obtains $q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ q'\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ (q,\ q')\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$|\ p\ p'$ **where** $q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ q'\ |\in|\ gtt\text{-}states\ \mathcal{G}_2\ q = p \vee (q,\ p)\ |\in|\ (eps\ (fst\ \mathcal{G}_1))|^+|$
$(p,\ p')\ |\in|\ \Delta_\varepsilon\ (snd\ \mathcal{G}_1)\ (fst\ \mathcal{G}_2)\ p' = q' \vee (p',\ q')\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+|$
$|\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_2\ (q,\ q')\ |\in|\ (eps\ (fst\ \mathcal{G}_2))|^+| \wedge q'\ |\in|\ gtt\text{-}states\ \mathcal{G}_2$
$\langle proof \rangle$

lemma $GTT\text{-}comp\text{-}eps\text{-}fst\text{-}statesD$:
$(p,\ q)\ |\in|\ eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)) \Longrightarrow p\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|\ gtt\text{-}states\ \mathcal{G}_2$
$(p,\ q)\ |\in|\ eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)) \Longrightarrow q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|\ gtt\text{-}states\ \mathcal{G}_2$
$\langle proof \rangle$

lemma $GTT\text{-}comp\text{-}eps\text{-}ftrancl\text{-}fst\text{-}statesD$:
$(p,\ q)\ |\in|\ (eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+| \Longrightarrow p\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|\ gtt\text{-}states\ \mathcal{G}_2$
$(p,\ q)\ |\in|\ (eps\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2)))|^+| \Longrightarrow q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1\ |\cup|\ gtt\text{-}states\ \mathcal{G}_2$
$\langle proof \rangle$

lemma $GTT\text{-}comp\text{-}first$:
assumes $q\ |\in|\ ta\text{-}der\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2))\ t\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1$
$gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
shows $q\ |\in|\ ta\text{-}der\ (fst\ \mathcal{G}_1)\ t$
$\langle proof \rangle$

lemma $GTT\text{-}comp\text{-}second$:
assumes $gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_2$
$q\ |\in|\ ta\text{-}der\ (snd\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2))\ t$
shows $q\ |\in|\ ta\text{-}der\ (snd\ \mathcal{G}_2)\ t$
$\langle proof \rangle$

lemma $gtt\text{-}comp\text{-}sound\text{-}semi$:
fixes $\mathcal{G}_1\ \mathcal{G}_2 :: ('f,\ 'q)\ gtt$
assumes $as2$: $gtt\text{-}states\ \mathcal{G}_1\ |\cap|\ gtt\text{-}states\ \mathcal{G}_2 = \{||\}$
and 1: $q\ |\in|\ gta\text{-}der\ (fst\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2))\ s\ q\ |\in|\ gta\text{-}der\ (snd\ (GTT\text{-}comp\ \mathcal{G}_1\ \mathcal{G}_2))\ t\ q\ |\in|\ gtt\text{-}states\ \mathcal{G}_1$
shows $\exists\, u.\ q\ |\in|\ gta\text{-}der\ (snd\ \mathcal{G}_1)\ u \wedge gtt\text{-}accept\ \mathcal{G}_2\ u\ t\ \langle proof \rangle$

lemma *gtt-comp-asound*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) ⊆ *gcomp-rel UNIV* (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$)
⟨*proof*⟩

lemma *gtt-comp-lang-complete*:
shows *gtt-lang* $\mathcal{G}_1$ *O gtt-lang* $\mathcal{G}_2$ ⊆ *gtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$)
⟨*proof*⟩

lemma *gtt-comp-alang*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gcomp-rel UNIV* (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$)
⟨*proof*⟩

lemma *gtt-comp-lang*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *gtt-lang* (*GTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gtt-lang* $\mathcal{G}_1$ *O gtt-lang* $\mathcal{G}_2$
⟨*proof*⟩

abbreviation *GTT-comp′* **where**
GTT-comp′ $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *GTT-comp* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *gtt-comp′-alang*:
shows *agtt-lang* (*GTT-comp′* $\mathcal{G}_1$ $\mathcal{G}_2$) = *gcomp-rel UNIV* (*agtt-lang* $\mathcal{G}_1$) (*agtt-lang* $\mathcal{G}_2$)
⟨*proof*⟩

end
theory *GTT-Transitive-Closure*
imports *GTT-Compose*
begin

4.8 GTT closure under transitivity

inductive-set Δ*-trancl-set* :: (′*q*, ′*f*) *ta* ⇒ (′*q*, ′*f*) *ta* ⇒ (′*q* × ′*q*) *set* **for** *A B*
where
Δ*-set-cong*: *TA-rule f ps p* |∈| *rules A* ⟹ *TA-rule f qs q* |∈| *rules B* ⟹ *length ps* = *length qs* ⟹
(⋀*i*. *i* < *length qs* ⟹ (*ps* ! *i*, *qs* ! *i*) ∈ Δ*-trancl-set A B*) ⟹ (*p*, *q*) ∈ Δ*-trancl-set A B*
| Δ*-set-eps1*: (*p*, *p′*) |∈| *eps A* ⟹ (*p*, *q*) ∈ Δ*-trancl-set A B* ⟹ (*p′*, *q*) ∈ Δ*-trancl-set A B*
| Δ*-set-eps2*: (*q*, *q′*) |∈| *eps B* ⟹ (*p*, *q*) ∈ Δ*-trancl-set A B* ⟹ (*p*, *q′*) ∈ Δ*-trancl-set A B*
| Δ*-set-trans*: (*p*, *q*) ∈ Δ*-trancl-set A B* ⟹ (*q*, *r*) ∈ Δ*-trancl-set A B* ⟹ (*p*, *r*) ∈ Δ*-trancl-set A B*

lemma Δ*-trancl-set-states*: Δ*-trancl-set* $\mathcal{A}$ $\mathcal{B}$ $\subseteq$ *fset* ($\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathcal{B}$)
⟨*proof*⟩

lemma *finite-*Δ*-trancl-set* [*simp*]: *finite* (Δ*-trancl-set* $\mathcal{A}$ $\mathcal{B}$)
⟨*proof*⟩

context
includes *fset.lifting*
begin
lift-definition Δ*-trancl* :: ($'q$, $'f$) *ta* $\Rightarrow$ ($'q$, $'f$) *ta* $\Rightarrow$ ($'q \times 'q$) *fset* **is** Δ*-trancl-set*
⟨*proof*⟩
lemmas Δ*-trancl-cong* = Δ*-set-cong* [*Transfer.transferred*]
lemmas Δ*-trancl-eps1* = Δ*-set-eps1* [*Transfer.transferred*]
lemmas Δ*-trancl-eps2* = Δ*-set-eps2* [*Transfer.transferred*]
lemmas Δ*-trancl-cases* = Δ*-trancl-set.cases*[*Transfer.transferred*]
lemmas Δ*-trancl-induct* [*consumes 1*, *case-names* Δ*-cong* Δ*-eps1* Δ*-eps2* Δ*-trans*]
= Δ*-trancl-set.induct*[*Transfer.transferred*]
lemmas Δ*-trancl-intros* = Δ*-trancl-set.intros*[*Transfer.transferred*]
lemmas Δ*-trancl-simps* = Δ*-trancl-set.simps*[*Transfer.transferred*]
end

lemma Δ*-trancl-cl* [*simp*]:
(Δ*-trancl* A B)$|^+|$ = Δ*-trancl* A B
⟨*proof*⟩

lemma Δ*-trancl-states*: Δ*-trancl* $\mathcal{A}$ $\mathcal{B}$ $|\subseteq|$ ($\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathcal{B}$)
⟨*proof*⟩

definition *GTT-trancl* **where**
GTT-trancl G =
(*let* Δ = Δ*-trancl* (*snd* G) (*fst* G) *in*
(*TA* (*rules* (*fst* G)) (*eps* (*fst* G) $|\cup|$ Δ),
TA (*rules* (*snd* G)) (*eps* (*snd* G) $|\cup|$ ($\Delta|^{-1}|$))))

lemma Δ*-trancl-inv*:
(Δ*-trancl* A B)$|^{-1}|$ = Δ*-trancl* B A
⟨*proof*⟩

lemma *gtt-states-GTT-trancl*:
gtt-states (*GTT-trancl* G) $|\subseteq|$ *gtt-states* G
⟨*proof*⟩

lemma *gtt-syms-GTT-trancl*:
gtt-syms (*GTT-trancl* G) = *gtt-syms* G
⟨*proof*⟩

lemma *GTT-trancl-base*:
gtt-lang G $\subseteq$ *gtt-lang* (*GTT-trancl* G)

⟨*proof*⟩

lemma *GTT-trancl-trans*:
gtt-lang (*GTT-comp* (*GTT-trancl G*) (*GTT-trancl G*)) ⊆ *gtt-lang* (*GTT-trancl G*)
⟨*proof*⟩

lemma *agtt-lang-base*:
agtt-lang G ⊆ *agtt-lang* (*GTT-trancl G*)
⟨*proof*⟩

lemma Δ_ε*-tr-incl*:
Δ_ε (*TA* (*rules A*) (*eps A* |∪| Δ*-trancl B A*)) (*TA* (*rules B*) (*eps B* |∪| Δ*-trancl A B*)) = Δ*-trancl A B*
(**is** *?LS* = *?RS*)
⟨*proof*⟩

lemma *agtt-lang-trans*:
gcomp-rel UNIV (*agtt-lang* (*GTT-trancl G*)) (*agtt-lang* (*GTT-trancl G*)) ⊆ *agtt-lang* (*GTT-trancl G*)
⟨*proof*⟩

lemma *GTT-trancl-acomplete*:
gtrancl-rel UNIV (*agtt-lang G*) ⊆ *agtt-lang* (*GTT-trancl G*)
⟨*proof*⟩

lemma *Restr-rtrancl-gtt-lang-eq-trancl-gtt-lang*:
$(gtt\text{-}lang\ G)^* = (gtt\text{-}lang\ G)^+$
⟨*proof*⟩

lemma *GTT-trancl-complete*:
$(gtt\text{-}lang\ G)^+$ ⊆ *gtt-lang* (*GTT-trancl G*)
⟨*proof*⟩

lemma *trancl-gtt-lang-arg-closed*:
assumes *length ss* = *length ts* ∀ *i* < *length ts*. (*ss* ! *i*, *ts* ! *i*) ∈ $(gtt\text{-}lang\ \mathcal{G})^+$
shows (*GFun f ss*, *GFun f ts*) ∈ $(gtt\text{-}lang\ \mathcal{G})^+$ (**is** *?e* ∈ -)
⟨*proof*⟩

lemma Δ*-trancl-sound*:
assumes (*p*, *q*) |∈| Δ*-trancl A B*
obtains *s t* **where** (*s*, *t*) ∈ $(gtt\text{-}lang\ (B, A))^+$ *p* |∈| *gta-der A s q* |∈| *gta-der B t*
⟨*proof*⟩

lemma *GTT-trancl-sound-aux*:
assumes *p* |∈| *gta-der* (*TA* (*rules A*) (*eps A* |∪| (Δ*-trancl B A*))) *s*
shows ∃ *t*. (*s*, *t*) ∈ $(gtt\text{-}lang\ (A, B))^+$ ∧ *p* |∈| *gta-der A t*

⟨*proof*⟩

lemma *GTT-trancl-asound*:
agtt-lang (*GTT-trancl G*) ⊆ *gtrancl-rel UNIV* (*agtt-lang G*)
⟨*proof*⟩

lemma *GTT-trancl-sound*:
gtt-lang (*GTT-trancl G*) ⊆ (*gtt-lang G*)$^+$
⟨*proof*⟩

lemma *GTT-trancl-alang*:
agtt-lang (*GTT-trancl G*) = *gtrancl-rel UNIV* (*agtt-lang G*)
⟨*proof*⟩

lemma *GTT-trancl-lang*:
gtt-lang (*GTT-trancl G*) = (*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*) *gtt* ⇒ (′*q* × ′*q*) *fset* ⇒ ′*f gterm rel* **where**
pair-at-lang 𝒢 *Q* = {(*s*, *t*) | *s t p q*. *q* |∈| *gta-der* (*fst* 𝒢) *s* ∧ *p* |∈| *gta-der* (*snd* 𝒢) *t* ∧ (*q*, *p*) |∈| *Q*}

lemma *pair-at-lang-restr-states*:
pair-at-lang 𝒢 *Q* = *pair-at-lang* 𝒢 (*Q* |∩| (𝒬 (*fst* 𝒢) |×| 𝒬 (*snd* 𝒢)))
⟨*proof*⟩

lemma *pair-at-langE*:
assumes (*s*, *t*) ∈ *pair-at-lang* 𝒢 *Q*
obtains *q p* **where** (*q*, *p*) |∈| *Q* **and** *q* |∈| *gta-der* (*fst* 𝒢) *s* **and** *p* |∈| *gta-der* (*snd* 𝒢) *t*
⟨*proof*⟩

lemma *pair-at-langI*:
assumes *q* |∈| *gta-der* (*fst* 𝒢) *s p* |∈| *gta-der* (*snd* 𝒢) *t* (*q*, *p*) |∈| *Q*
shows (*s*, *t*) ∈ *pair-at-lang* 𝒢 *Q*
⟨*proof*⟩

lemma *pair-at-lang-fun-states*:
assumes *finj-on f* (𝒬 (*fst* 𝒢)) **and** *finj-on g* (𝒬 (*snd* 𝒢))
and *Q* |⊆| 𝒬 (*fst* 𝒢) |×| 𝒬 (*snd* 𝒢)
shows *pair-at-lang* 𝒢 *Q* = *pair-at-lang* (*map-prod* (*fmap-states-ta f*) (*fmap-states-ta g*) 𝒢) (*map-prod f g* |‘| *Q*)

(**is** *?LS* = *?RS*)
⟨*proof*⟩

lemma *converse-pair-at-lang*:
$(\mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}\ Q)^{-1} = \mathit{pair\text{-}at\text{-}lang}\ (\mathit{prod.swap}\ \mathcal{G})\ (Q|^{-1}|)$
⟨*proof*⟩

lemma *pair-at-agtt*:
$\mathit{agtt\text{-}lang}\ \mathcal{G} = \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}\ (\mathit{fId\text{-}on}\ (\mathit{gtt\text{-}interface}\ \mathcal{G}))$
⟨*proof*⟩

definition Δ-*eps-pair* **where**
$\Delta\text{-}\mathit{eps\text{-}pair}\ \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2 \equiv\ Q_1\ |O|\ \Delta_\varepsilon\ (\mathit{snd}\ \mathcal{G}_1)\ (\mathit{fst}\ \mathcal{G}_2)\ |O|\ Q_2$

lemma *pair-comp-sound1*:
assumes $(s,\ t) \in \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_1\ Q_1$
and $(t,\ u) \in \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_2\ Q_2$
shows $(s,\ u) \in \mathit{pair\text{-}at\text{-}lang}\ (\mathit{fst}\ \mathcal{G}_1,\ \mathit{snd}\ \mathcal{G}_2)\ (\Delta\text{-}\mathit{eps\text{-}pair}\ \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2)$
⟨*proof*⟩

lemma *pair-comp-sound2*:
assumes $(s,\ u) \in\ \mathit{pair\text{-}at\text{-}lang}\ (\mathit{fst}\ \mathcal{G}_1,\ \mathit{snd}\ \mathcal{G}_2)\ (\Delta\text{-}\mathit{eps\text{-}pair}\ \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2)$
shows $\exists\ t.\ (s,\ t) \in \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_1\ Q_1 \wedge (t,\ u) \in \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_2\ Q_2$
⟨*proof*⟩

lemma *pair-comp-sound*:
$\mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_1\ Q_1\ O\ \mathit{pair\text{-}at\text{-}lang}\ \mathcal{G}_2\ Q_2 = \mathit{pair\text{-}at\text{-}lang}\ (\mathit{fst}\ \mathcal{G}_1,\ \mathit{snd}\ \mathcal{G}_2)\ (\Delta\text{-}\mathit{eps\text{-}pair}\ \mathcal{G}_1\ Q_1\ \mathcal{G}_2\ Q_2)$
⟨*proof*⟩

inductive-set Δ-*Atrans-set* :: $('q \times 'q)$ *fset* ⇒ $('q,\ 'f)$ *ta* ⇒ $('q,\ 'f)$ *ta* ⇒ $('q \times 'q)$ *set* **for** $Q\ \mathcal{A}\ \mathcal{B}$ **where**
base [*simp*]: $(p,\ q)\ |\in|\ Q \Longrightarrow (p,\ q) \in \Delta\text{-}\mathit{Atrans\text{-}set}\ Q\ \mathcal{A}\ \mathcal{B}$
| *step* [*intro*]: $(p,\ q) \in \Delta\text{-}\mathit{Atrans\text{-}set}\ Q\ \mathcal{A}\ \mathcal{B} \Longrightarrow (q,\ r)\ |\in|\ \Delta_\varepsilon\ \mathcal{B}\ \mathcal{A} \Longrightarrow$
$(r,\ v) \in \Delta\text{-}\mathit{Atrans\text{-}set}\ Q\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p,\ v) \in \Delta\text{-}\mathit{Atrans\text{-}set}\ Q\ \mathcal{A}\ \mathcal{B}$

lemma Δ-*Atrans-set-states*:
$(p,\ q) \in \Delta\text{-}\mathit{Atrans\text{-}set}\ Q\ \mathcal{A}\ \mathcal{B} \Longrightarrow (p,\ q) \in \mathit{fset}\ ((\mathit{fst}\ |`|\ Q\ |\cup|\ \mathcal{Q}\ \mathcal{A})\ |\times|\ (\mathit{snd}\ |`|\ Q\ |\cup|\ \mathcal{Q}\ \mathcal{B}))$
⟨*proof*⟩

lemma *finite*-Δ-*Atrans-set*: *finite* (Δ-*Atrans-set* $Q\ \mathcal{A}\ \mathcal{B}$)
⟨*proof*⟩

context
includes *fset.lifting*
begin
lift-definition Δ-*Atrans* :: $('q \times 'q)$ *fset* ⇒ $('q,\ 'f)$ *ta* ⇒ $('q,\ 'f)$ *ta* ⇒ $('q \times 'q)$ *fset* **is** Δ-*Atrans-set*

⟨*proof*⟩

lemmas Δ-*Atrans-base* [*simp*] = Δ-*Atrans-set.base* [*Transfer.transferred*]
lemmas Δ-*Atrans-step* [*intro*] = Δ-*Atrans-set.step* [*Transfer.transferred*]
lemmas Δ-*Atrans-cases* = Δ-*Atrans-set.cases*[*Transfer.transferred*]
lemmas Δ-*Atrans-induct* [*consumes 1*, *case-names base step*] = Δ-*Atrans-set.induct*[*Transfer.transferred*]
end

abbreviation Δ-*Atrans-gtt* $\mathcal{G}$ $Q \equiv \Delta$-*Atrans* Q (*fst* $\mathcal{G}$) (*snd* $\mathcal{G}$)

lemma *pair-trancl-sound1*:
assumes $(s, t) \in$ (*pair-at-lang* $\mathcal{G}$ $Q)^+$
shows $\exists$ q p. p $|\in|$ *gta-der* (*fst* $\mathcal{G}$) $s \wedge q$ $|\in|$ *gta-der* (*snd* $\mathcal{G}$) $t \wedge (p, q)$ $|\in|$ Δ-*Atrans-gtt* $\mathcal{G}$ Q
⟨*proof*⟩

lemma *pair-trancl-sound2*:
assumes (p, q) $|\in|$ Δ-*Atrans-gtt* $\mathcal{G}$ Q
and p $|\in|$ *gta-der* (*fst* $\mathcal{G}$) s q $|\in|$ *gta-der* (*snd* $\mathcal{G}$) t
shows $(s, t) \in$ (*pair-at-lang* $\mathcal{G}$ $Q)^+$ ⟨*proof*⟩

lemma *pair-trancl-sound*:
(*pair-at-lang* $\mathcal{G}$ $Q)^+$ = *pair-at-lang* $\mathcal{G}$ (Δ-*Atrans-gtt* $\mathcal{G}$ Q)
⟨*proof*⟩

abbreviation *fst-pair-cl* $\mathcal{A}$ $Q \equiv$ *TA* (*rules* $\mathcal{A}$) (*eps* $\mathcal{A}$ $|\cup|$ (*fId-on* ($\mathcal{Q}$ $\mathcal{A}$) $|O|$ Q))
definition *pair-at-to-agtt* :: $('q, 'f)$ *gtt* $\Rightarrow$ $('q \times 'q)$ *fset* $\Rightarrow$ $('q, 'f)$ *gtt* **where**
pair-at-to-agtt $\mathcal{G}$ Q = (*fst-pair-cl* (*fst* $\mathcal{G}$) Q , *TA* (*rules* (*snd* $\mathcal{G}$)) (*eps* (*snd* $\mathcal{G}$)))

lemma *fst-pair-cl-eps*:
assumes (p, q) $|\in|$ (*eps* (*fst-pair-cl* $\mathcal{A}$ Q))$|^+|$
and $\mathcal{Q}$ $\mathcal{A}$ $|\cap|$ *snd* |'| Q = {||}
shows (p, q) $|\in|$ (*eps* $\mathcal{A}$)$|^+|$ $\vee$ ($\exists$ r. ($p = r \vee (p, r)$ $|\in|$ (*eps* $\mathcal{A}$)$|^+|$) $\wedge$ (r, q) $|\in|$ Q) ⟨*proof*⟩

lemma *fst-pair-cl-res-aux*:
assumes $\mathcal{Q}$ $\mathcal{A}$ $|\cap|$ *snd* |'| Q = {||}
and q $|\in|$ *ta-der* (*fst-pair-cl* $\mathcal{A}$ Q) (*term-of-gterm* t)
shows $\exists$ p. p $|\in|$ *ta-der* $\mathcal{A}$ (*term-of-gterm* t) $\wedge$ (q $|\notin|$ $\mathcal{Q}$ $\mathcal{A}$ $\longrightarrow$ (p, q) $|\in|$ Q) $\wedge$ (q $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ $\longrightarrow$ $p = q$) ⟨*proof*⟩

lemma *restr-distjoing*:
assumes Q $|\subseteq|$ $\mathcal{Q}$ $\mathcal{A}$ $|\times|$ $\mathcal{Q}$ $\mathfrak{B}$
and $\mathcal{Q}$ $\mathcal{A}$ $|\cap|$ $\mathcal{Q}$ $\mathfrak{B}$ = {||}
shows $\mathcal{Q}$ $\mathcal{A}$ $|\cap|$ *snd* |'| Q = {||}
⟨*proof*⟩

lemma *pair-at-agtt-conv*:
assumes Q $|\subseteq|$ $\mathcal{Q}$ (*fst* $\mathcal{G}$) $|\times|$ $\mathcal{Q}$ (*snd* $\mathcal{G}$) **and** $\mathcal{Q}$ (*fst* $\mathcal{G}$) $|\cap|$ $\mathcal{Q}$ (*snd* $\mathcal{G}$) = {||}

shows *pair-at-lang* $\mathcal{G}$ *Q* = *agtt-lang* (*pair-at-to-agtt* $\mathcal{G}$ *Q*) (**is** *?LS* = *?RS*)
⟨*proof*⟩

definition *pair-at-to-agtt′* **where**
pair-at-to-agtt′ $\mathcal{G}$ *Q* = (*let* $\mathcal{A}$ = *fmap-states-ta Inl* (*fst* $\mathcal{G}$) *in*
let $\mathcal{B}$ = *fmap-states-ta Inr* (*snd* $\mathcal{G}$) *in*
let *Q′* = *Q* |∩| ($\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)) *in*
pair-at-to-agtt ($\mathcal{A}$, $\mathcal{B}$) (*map-prod Inl Inr* |'| *Q′*))

lemma *pair-at-agtt-cost*:
pair-at-lang $\mathcal{G}$ *Q* = *agtt-lang* (*pair-at-to-agtt′* $\mathcal{G}$ *Q*)
⟨*proof*⟩

lemma Δ-*Atrans-states-stable*:
assumes *Q* |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
shows Δ-*Atrans-gtt* $\mathcal{G}$ *Q* |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
⟨*proof*⟩

lemma Δ-*Atrans-map-prod*:
assumes *finj-on f* ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) **and** *finj-on g* ($\mathcal{Q}$ (*snd* $\mathcal{G}$))
and *Q* |⊆| $\mathcal{Q}$ (*fst* $\mathcal{G}$) |×| $\mathcal{Q}$ (*snd* $\mathcal{G}$)
shows *map-prod f g* |'| (Δ-*Atrans-gtt* $\mathcal{G}$ *Q*) = Δ-*Atrans-gtt* (*map-prod* (*fmap-states-ta f*) (*fmap-states-ta g*) $\mathcal{G}$) (*map-prod f g* |'| *Q*)
(**is** *?LS* = *?RS*)
⟨*proof*⟩

definition *Q-pow* **where**
Q-pow Q $\mathcal{S}_1$ $\mathcal{S}_2$ =
{|(*Wrapp X*, *Wrapp Y*) | *X Y p q*. *X* |∈| *fPow* $\mathcal{S}_1$ ∧ *Y* |∈| *fPow* $\mathcal{S}_2$ ∧ *p* |∈| *X* ∧ *q* |∈| *Y* ∧ (*p*, *q*) |∈| *Q*|}

lemma *Q-pow-fmember*:
(*X*, *Y*) |∈| *Q-pow Q* $\mathcal{S}_1$ $\mathcal{S}_2$ ⟷ (∃ *p q*. *ex X* |∈| *fPow* $\mathcal{S}_1$ ∧ *ex Y* |∈| *fPow* $\mathcal{S}_2$ ∧ *p* |∈| *ex X* ∧ *q* |∈| *ex Y* ∧ (*p*, *q*) |∈| *Q*)
⟨*proof*⟩

lemma *pair-automaton-det-lang-sound-complete*:
pair-at-lang $\mathcal{G}$ *Q* = *pair-at-lang* (*map-both ps-ta* $\mathcal{G}$) (*Q-pow Q* ($\mathcal{Q}$ (*fst* $\mathcal{G}$)) ($\mathcal{Q}$ (*snd* $\mathcal{G}$))) (**is** *?LS* = *?RS*)
⟨*proof*⟩

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* ($\mathcal{A}$, $\mathcal{B}$) ($\mathcal{Q}$ $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$ |−| *Q*) = *gterms* (*fset* $\mathcal{F}$) × *gterms* (*fset* $\mathcal{F}$) − *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 $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ (*ta-union* (*fst* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$),
          ta-union (*snd* $\mathcal{G}_1$) (*snd* $\mathcal{G}_2$))

abbreviation *AGTT-union′* **where**
  AGTT-union′ $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *AGTT-union* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *disj-gtt-states-disj-fst-ta-states*:
  assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
  shows $\mathcal{Q}$ (*fst* $\mathcal{G}_1$) |∩| $\mathcal{Q}$ (*fst* $\mathcal{G}_2$) = {||}
  ⟨*proof*⟩

lemma *disj-gtt-states-disj-snd-ta-states*:
  assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
  shows $\mathcal{Q}$ (*snd* $\mathcal{G}_1$) |∩| $\mathcal{Q}$ (*snd* $\mathcal{G}_2$) = {||}
  ⟨*proof*⟩

lemma *ta-der-not-contains-undefined-state*:
  assumes *q* |∉| $\mathcal{Q}$ *T* **and** *ground t*
  shows *q* |∉| *ta-der T t*
  ⟨*proof*⟩

lemma *AGTT-union-sound1*:
  assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
  shows *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$) ⊆ *agtt-lang* $\mathcal{G}_1$ ∪ *agtt-lang* $\mathcal{G}_2$
⟨*proof*⟩

lemma *AGTT-union-sound2*:
  shows *agtt-lang* $\mathcal{G}_1$ ⊆ *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$)
    agtt-lang $\mathcal{G}_2$ ⊆ *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$)
  ⟨*proof*⟩

lemma *AGTT-union-sound*:
  assumes *dist-st*: *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
  shows *agtt-lang* (*AGTT-union* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ ∪ *agtt-lang* $\mathcal{G}_2$
  ⟨*proof*⟩

lemma *AGTT-union′-sound*:
  fixes $\mathcal{G}_1$ :: (′*q*, ′*f*) *gtt* **and** $\mathcal{G}_2$ :: (′*q*, ′*f*) *gtt*
  shows *agtt-lang* (*AGTT-union′* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ ∪ *agtt-lang* $\mathcal{G}_2$
⟨*proof*⟩

4.10 Anchord gtt compositon

definition *AGTT-comp* :: $('q, 'f)$ *gtt* ⇒ $('q, 'f)$ *gtt* ⇒ $('q, 'f)$ *gtt* **where**
AGTT-comp $\mathcal{G}_1$ $\mathcal{G}_2$ = (*let* $(\mathcal{A}, \mathcal{B})$ = (*fst* $\mathcal{G}_1$, *snd* $\mathcal{G}_2$) *in*
(*TA* (*rules* $\mathcal{A}$) (*eps* $\mathcal{A}$ |∪| (Δ_ε (*snd* $\mathcal{G}_1$) (*fst* $\mathcal{G}_2$) |∩| (*gtt-interface* $\mathcal{G}_1$ |×| *gtt-interface* $\mathcal{G}_2$))),
TA (*rules* $\mathcal{B}$) (*eps* $\mathcal{B}$)))

abbreviation *AGTT-comp′* **where**
AGTT-comp′ $\mathcal{G}_1$ $\mathcal{G}_2$ ≡ *AGTT-comp* (*fmap-states-gtt Inl* $\mathcal{G}_1$) (*fmap-states-gtt Inr* $\mathcal{G}_2$)

lemma *AGTT-comp-sound*:
assumes *gtt-states* $\mathcal{G}_1$ |∩| *gtt-states* $\mathcal{G}_2$ = {||}
shows *agtt-lang* (*AGTT-comp* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ *O agtt-lang* $\mathcal{G}_2$
⟨*proof*⟩

lemma *AGTT-comp′-sound*:
agtt-lang (*AGTT-comp′* $\mathcal{G}_1$ $\mathcal{G}_2$) = *agtt-lang* $\mathcal{G}_1$ *O agtt-lang* $\mathcal{G}_2$
⟨*proof*⟩

4.11 Anchord gtt transitivity

definition *AGTT-trancl* :: $('q, 'f)$ *gtt* ⇒ $('q + 'q, 'f)$ *gtt* **where**
AGTT-trancl $\mathcal{G}$ = (*let* $\mathcal{A}$ = *fmap-states-ta Inl* (*fst* $\mathcal{G}$) *in*
(*TA* (*rules* $\mathcal{A}$) (*eps* $\mathcal{A}$ |∪| *map-prod CInl CInr* |‘| (Δ-*Atrans-gtt* $\mathcal{G}$ (*fId-on* (*gtt-interface* $\mathcal{G}$)))),
TA (*map-ta-rule CInr id* |‘| (*rules* (*snd* $\mathcal{G}$))) (*map-both CInr* |‘| (*eps* (*snd* $\mathcal{G}$)))))

lemma *AGTT-trancl-sound*:
shows *agtt-lang* (*AGTT-trancl* $\mathcal{G}$) = (*agtt-lang* $\mathcal{G}$)$^+$
⟨*proof*⟩

4.12 Anchord gtt triming

abbreviation *trim-agtt* ≡ *trim-gtt*

lemma *agtt-only-prod-lang*:
agtt-lang (*gtt-only-prod* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$ (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *agtt-only-reach-lang*:
agtt-lang (*gtt-only-reach* $\mathcal{G}$) = *agtt-lang* $\mathcal{G}$
⟨*proof*⟩

lemma *trim-agtt-lang* [*simp*]:
agtt-lang (*trim-agtt G*) = *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 gterm* ⇒ *'g gterm* ⇒ (*'f option* × *'g option*) *gterm* **where**
gpair s t = *glabel* (λ*p*. (*gfun-at s p*, *gfun-at t p*)) (*gunion* (*gdomain s*) (*gdomain t*))

We provide an efficient implementation of gpair.

definition *zip-fill* :: *'a list* ⇒ *'b list* ⇒ (*'a option* × *'b option*) *list* **where**
zip-fill xs ys = *zip* (*map Some xs* @ *replicate* (*length ys* − *length xs*) *None*)
(*map Some ys* @ *replicate* (*length xs* − *length ys*) *None*)

lemma *zip-fill-code* [*code*]:
zip-fill xs [] = *map* (λ*x*. (*Some x*, *None*)) *xs*
zip-fill [] *ys* = *map* (λ*y*. (*None*, *Some y*)) *ys*
zip-fill (*x* # *xs*) (*y* # *ys*) = (*Some x*, *Some y*) # *zip-fill xs ys*
⟨*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* (λ*f*. (*Some f*, *None*)) *s*
| *gpair-impl None* (*Some t*) = *map-gterm* (λ*f*. (*None*, *Some f*)) *t*
| *gpair-impl None None* = *GFun* (*None*, *None*) []

declare *gpair-impl.simps*(*2*−*4*)[*code*]

lemma *gpair-impl-code* [*simp*, *code*]:
gpair-impl (*Some s*) (*Some t*) =

$(case\ s\ of\ GFun\ f\ ss \Rightarrow case\ t\ of\ GFun\ g\ ts \Rightarrow$
$GFun\ (Some\ f,\ Some\ g)\ (map\ (\lambda(s,\ t).\ gpair\text{-}impl\ s\ t)\ (zip\text{-}fill\ ss\ ts)))$
⟨*proof*⟩

lemma *gpair-code* [*code*]:
$gpair\ s\ t = gpair\text{-}impl\ (Some\ s)\ (Some\ t)$
⟨*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*:
$map\text{-}gterm\ prod.swap\ (gpair\ s\ t) = gpair\ t\ s$
⟨*proof*⟩

lemma *gpair-assoc*:
defines $f \equiv \lambda(f,\ gh).\ (f,\ gh \gg= fst,\ gh \gg= snd)$
defines $g \equiv \lambda(fg,\ h).\ (fg \gg= fst,\ fg \gg= snd,\ h)$
shows $map\text{-}gterm\ f\ (gpair\ s\ (gpair\ t\ u)) = map\text{-}gterm\ g\ (gpair\ (gpair\ s\ t)\ u)$
⟨*proof*⟩

5.2 Decoding of pairs

fun $gcollapse :: 'f\ option\ gterm \Rightarrow 'f\ gterm\ option$ **where**
$gcollapse\ (GFun\ None\ \text{-}) = None$
$|\ gcollapse\ (GFun\ (Some\ f)\ ts) = Some\ (GFun\ f\ (map\ the\ (filter\ (\lambda t.\ \neg\ Option.is\text{-}none\ t)\ (map\ gcollapse\ ts))))$

lemma *gcollapse-groot-None* [*simp*]:
$groot\text{-}sym\ t = None \Longrightarrow gcollapse\ t = None$
$fst\ (groot\ t) = None \Longrightarrow gcollapse\ t = None$
⟨*proof*⟩

definition $gfst :: ('f\ option \times 'g\ option)\ gterm \Rightarrow 'f\ gterm$ **where**
$gfst = the \circ gcollapse \circ map\text{-}gterm\ fst$

definition $gsnd :: ('f\ option \times 'g\ option)\ gterm \Rightarrow 'g\ gterm$ **where**
$gsnd = the \circ gcollapse \circ map\text{-}gterm\ snd$

lemma *filter-less-upt*:
$[i \leftarrow [i..<m]\ .\ i < n] = [i..<min\ n\ m]$
⟨*proof*⟩

lemma *gcollapse-aux*:
assumes $gposs\ s = \{p.\ p \in gposs\ t \wedge gfun\text{-}at\ t\ p \neq Some\ None\}$

shows *gposs* (*the* (*gcollapse t*)) = *gposs s*
$\bigwedge p.\ p \in gposs\ s \Longrightarrow gfun\text{-}at\ (the\ (gcollapse\ t))\ p = (gfun\text{-}at\ t\ p \gg= id)$
⟨*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 $\bigwedge i.\ i < length\ ts \Longrightarrow ts\ !\ i = gpair\ (ss\ !\ i)\ (us\ !\ i)$
and *length ss* = *length ts* **and** *length us* = *length ts*
shows *GFun* (*Some f*, *Some h*) *ts* = *gpair* (*GFun f ss*) (*GFun h us*)
⟨*proof*⟩

lemma *map-funs-term-some-gpair*:
shows *gpair t t* = *map-gterm* (λ*f*. (*Some f*, *Some f*)) *t*
⟨*proof*⟩

lemma *gpair-inject* [*simp*]:
$gpair\ s\ t = gpair\ s'\ t' \longleftrightarrow s = s' \wedge t = t'$
⟨*proof*⟩

abbreviation *gterm-to-None-Some* :: *'f gterm* ⇒ (*'f option* × *'f option*) *gterm*
where
gterm-to-None-Some t ≡ *map-gterm* (λ*f*. (*None*, *Some f*)) *t*
abbreviation *gterm-to-Some-None t* ≡ *map-gterm* (λ*f*. (*Some f*, *None*)) *t*

lemma *inj-gterm-to-None-Some*: *inj gterm-to-None-Some*
⟨*proof*⟩

lemma *zip-fill1*:
assumes *length ss* < *length ts*

shows *zip-fill ss ts* = *zip* (*map Some ss*) (*map Some* (*take* (*length ss*) *ts*)) @
map (λ *x*. (*None*, *Some x*)) (*drop* (*length ss*) *ts*)
⟨*proof*⟩

lemma *zip-fill2*:
assumes *length ts* < *length ss*
shows *zip-fill ss ts* = *zip* (*map Some* (*take* (*length ts*) *ss*)) (*map Some ts*) @
map (λ *x*. (*Some x*, *None*)) (*drop* (*length ts*) *ss*)
⟨*proof*⟩

lemma *not-gposs-append* [*simp*]:
assumes $p \notin$ *gposs t*
shows *p* @ *q* ∈ *gposs t* = *False* ⟨*proof*⟩

lemma *gfun-at-gpair*:
gfun-at (*gpair s t*) *p* = (*if p* ∈ *gposs s then* (*if p* ∈ *gposs t*
then Some (*gfun-at s p*, *gfun-at t p*)
else Some (*gfun-at s p*, *None*)) *else*
(*if p* ∈ *gposs t then Some* (*None*, *gfun-at t p*) *else None*))
⟨*proof*⟩

lemma *gposs-of-gpair* [*simp*]:
shows *gposs* (*gpair s t*) = *gposs s* ∪ *gposs t*
⟨*proof*⟩

lemma *poss-to-gpair-poss*:
p ∈ *gposs s* ⟹ *p* ∈ *gposs* (*gpair s t*)
p ∈ *gposs t* ⟹ *p* ∈ *gposs* (*gpair s t*)
⟨*proof*⟩

lemma *gsubt-at-gpair-poss*:
assumes *p* ∈ *gposs s* **and** *p* ∈ *gposs t*
shows *gsubt-at* (*gpair s t*) *p* = *gpair* (*gsubt-at s p*) (*gsubt-at t p*) ⟨*proof*⟩

lemma *subst-at-gpair-nt-poss-Some-None*:
assumes *p* ∈ *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* ∈ *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* ⇒ *'f option list gterm* **where**
gencode ts = *glabel* (λ*p. map* (λ*t. gfun-at t p*) *ts*) (*gunions* (*map gdomain ts*))

definition *gdecode-nth* :: *'f option list gterm* ⇒ *nat* ⇒ *'f gterm* **where**
gdecode-nth t i = *the* (*gcollapse* (*map-gterm* (λ*f. f* ! *i*) *t*))

lemma *gdecode-nth-gencode*:
assumes $i <$ *length ts*
shows *gdecode-nth* (*gencode ts*) *i* = *ts* ! *i*
⟨*proof*⟩

definition *gdecode* :: *'f option list gterm* ⇒ *'f gterm list* **where**
gdecode t = (*case t of GFun f ts* ⇒ *map* (λ*i. gdecode-nth t i*) [*0..<length f*])

lemma *gdecode-gencode*:
gdecode (*gencode ts*) = *ts*
⟨*proof*⟩

definition *gencode-impl* :: *'f gterm option list* ⇒ *'f option list gterm* **where**
gencode-impl ts = *glabel* (λ*p. map* (λ*t. t* ≫= (λ*t. gfun-at t p*)) *ts*) (*gunions* (*map* (*case-option* (*GFun* () []) *gdomain*) *ts*))

lemma *gencode-code* [*code*]:
gencode ts = *gencode-impl* (*map Some ts*)
⟨*proof*⟩

lemma *gencode-singleton*:
gencode [*t*] = *map-gterm* (λ*f*. [*Some f*]) *t*
⟨*proof*⟩

lemma *gencode-pair*:
  gencode [*t*, *u*] = *map-gterm* (λ(*f*, *g*). [*f*, *g*]) (*gpair t u*)
  ⟨*proof*⟩

5.5 RRn relations

definition *RR1-spec* **where**
  RR1-spec A T ⟷ ℒ *A* = *T*

definition *RR2-spec* **where**
  RR2-spec A T ⟷ ℒ *A* = {*gpair t u* |*t u*. (*t*, *u*) ∈ *T*}

definition *RRn-spec* **where**
  RRn-spec n A R ⟷ ℒ *A* = *gencode* ‘ *R* ∧ (∀ *ts* ∈ *R*. *length ts* = *n*)

lemma *RR1-to-RRn-spec*:
  assumes *RR1-spec A T*
  shows *RRn-spec 1* (*fmap-funs-reg* (λ*f*. [*Some f*]) *A*) ((λ*t*. [*t*]) ‘ *T*)
⟨*proof*⟩

lemma *RR2-to-RRn-spec*:
  assumes *RR2-spec A T*
  shows *RRn-spec 2* (*fmap-funs-reg* (λ(*f*, *g*). [*f*, *g*]) *A*) ((λ(*t*, *u*). [*t*, *u*]) ‘ *T*)
⟨*proof*⟩

lemma *RRn-to-RR2-spec*:
  assumes *RRn-spec 2 A T*
  shows *RR2-spec* (*fmap-funs-reg* (λ *f*. (*f* ! *0* , *f* ! *1*)) *A*) ((λ *f*. (*f* ! *0*, *f* ! *1*)) ‘ *T*) (**is** *RR2-spec ?A ?T*)
⟨*proof*⟩

lemma *relabel-RR1-spec* [*simp*]:
  RR1-spec (*relabel-reg A*) *T* ⟷ *RR1-spec A T*
  ⟨*proof*⟩

lemma *relabel-RR2-spec* [*simp*]:
  RR2-spec (*relabel-reg A*) *T* ⟷ *RR2-spec A T*
  ⟨*proof*⟩

lemma *relabel-RRn-spec* [*simp*]:
  RRn-spec n (*relabel-reg A*) *T* ⟷ *RRn-spec n A T*
  ⟨*proof*⟩

lemma *trim-RR1-spec* [*simp*]:
  RR1-spec (*trim-reg A*) *T* ⟷ *RR1-spec A T*
  ⟨*proof*⟩

lemma *trim-RR2-spec* [*simp*]:

RR2-spec (*trim-reg A*) *T* ⟷ *RR2-spec A T*
⟨*proof*⟩

lemma *trim-RRn-spec* [*simp*]:
RRn-spec n (*trim-reg A*) *T* ⟷ *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* {|[] [] → *q*|} {||})) {[]}
⟨*proof*⟩

5.7 Pairing RR1 languages

cf. *gpair*.

abbreviation *lift-Some-None s* ≡ (*Some s*, *None*)
abbreviation *lift-None-Some s* ≡ (*None*, *Some s*)
abbreviation *pair-eps A B* ≡ (λ (*p*, *q*). ((*Some* (*fst p*), *q*), (*Some* (*snd p*), *q*))) |‘| (*eps A* |×| *finsert None* (*Some* |‘| 𝒬 *B*))
abbreviation *pair-rule* ≡ (λ (*ra*, *rb*). *TA-rule* (*Some* (*r-root ra*), *Some* (*r-root rb*)) (*zip-fill* (*r-lhs-states ra*) (*r-lhs-states rb*)) (*Some* (*r-rhs ra*), *Some* (*r-rhs rb*)))

lemma *lift-Some-None-pord-swap* [*simp*]:
prod.swap ∘ *lift-Some-None* = *lift-None-Some*
prod.swap ∘ *lift-None-Some* = *lift-Some-None*
⟨*proof*⟩

lemma *eps-to-pair-eps-Some-None*:
(*p*, *q*) |∈| *eps* 𝒜 ⟹ (*lift-Some-None p*, *lift-Some-None q*) |∈| *pair-eps* 𝒜 ℬ
⟨*proof*⟩

definition *pair-automaton* :: (′*p*, ′*f*) *ta* ⇒ (′*q*, ′*g*) *ta* ⇒ (′*p option* × ′*q option*, ′*f option* × ′*g option*) *ta* **where**
pair-automaton A B = *TA*
(*map-ta-rule lift-Some-None lift-Some-None* |‘| *rules A* |∪|
map-ta-rule lift-None-Some lift-None-Some |‘| *rules B* |∪|
pair-rule |‘| (*rules A* |×| *rules B*))
(*pair-eps A B* |∪| *map-both prod.swap* |‘| (*pair-eps B A*))

definition *pair-automaton-reg* **where**

pair-automaton-reg R L = *Reg* (*Some* |'| *fin R* |×| *Some* |'| *fin L*) (*pair-automaton* (*ta R*) (*ta L*))

lemma *pair-automaton-eps-simps*:
(*lift-Some-None p*, p') |∈| *eps* (*pair-automaton A B*) ⟷ (*lift-Some-None p*, p') |∈| *pair-eps A B*
(*q* , *lift-Some-None* q') |∈| *eps* (*pair-automaton A B*) ⟷ (*q* , *lift-Some-None* q') |∈| *pair-eps A B*
⟨*proof*⟩

lemma *pair-automaton-eps-Some-SomeD*:
((*Some p*, *Some* p'), *r*) |∈| *eps* (*pair-automaton A B*) ⟹ *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧ (*Some p* = *fst r* ∨ *Some* p' = *snd r*) ∧
(*Some p* ≠ *fst r* ⟶ (*p*, *the* (*fst r*)) |∈| (*eps A*)) ∧ (*Some* p' ≠ *snd r* ⟶ (p', *the* (*snd r*)) |∈| (*eps B*))
⟨*proof*⟩

lemma *pair-automaton-eps-Some-SomeD2*:
(*r*, (*Some p*, *Some* p')) |∈| *eps* (*pair-automaton A B*) ⟹ *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧ (*fst r* = *Some p* ∨ *snd r* = *Some* p') ∧
(*fst r* ≠ *Some p* ⟶ (*the* (*fst r*), *p*) |∈| (*eps A*)) ∧ (*snd r* ≠ *Some* p' ⟶ (*the* (*snd r*), p') |∈| (*eps B*))
⟨*proof*⟩

lemma *pair-eps-Some-None*:
fixes *p q* q'
defines *l* ≡ (*p*, *q*) **and** *r* ≡ *lift-Some-None* q'
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))$|^+|$
shows *q* = *None* ∧ *p* ≠ *None* ∧ (*the p*, q') |∈| (*eps A*)$|^+|$ ⟨*proof*⟩

lemma *pair-eps-Some-Some*:
fixes *p q*
defines *l* ≡ (*Some p*, *Some q*)
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))$|^+|$
shows *fst r* ≠ *None* ∧ *snd r* ≠ *None* ∧
(*fst l* ≠ *fst r* ⟶ (*p*, *the* (*fst r*)) |∈| (*eps A*)$|^+|$) ∧
(*snd l* ≠ *snd r* ⟶ (*q*, *the* (*snd r*)) |∈| (*eps B*)$|^+|$)
⟨*proof*⟩

lemma *pair-eps-Some-Some2*:
fixes *p q*
defines *r* ≡ (*Some p*, *Some q*)
assumes (*l*, *r*) |∈| (*eps* (*pair-automaton A B*))$|^+|$
shows *fst l* ≠ *None* ∧ *snd l* ≠ *None* ∧
(*fst l* ≠ *fst r* ⟶ (*the* (*fst l*), *p*) |∈| (*eps A*)$|^+|$) ∧
(*snd l* ≠ *snd r* ⟶ (*the* (*snd l*), *q*) |∈| (*eps B*)$|^+|$)
⟨*proof*⟩

lemma *map-pair-automaton*:
pair-automaton (*fmap-funs-ta f A*) (*fmap-funs-ta g B*) =
fmap-funs-ta (λ(*a*, *b*). (*map-option f a*, *map-option g b*)) (*pair-automaton A B*)
(**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemmas *map-pair-automaton-12* =
map-pair-automaton[*of - - id*, *unfolded fmap-funs-ta-id option.map-id*]
map-pair-automaton[*of id - -*, *unfolded fmap-funs-ta-id option.map-id*]

lemma *fmap-states-funs-ta-commute*:
fmap-states-ta f (*fmap-funs-ta g A*) = *fmap-funs-ta g* (*fmap-states-ta f A*)
⟨*proof*⟩

lemma *states-pair-automaton*:
𝒬 (*pair-automaton A B*) |⊆| (*finsert None* (*Some* |'| 𝒬 *A*) |×| (*finsert None* (*Some* |'| 𝒬 *B*)))
⟨*proof*⟩

lemma *swap-pair-automaton*:
assumes (*p*, *q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm t*)
shows (*q*, *p*) |∈| *ta-der* (*pair-automaton B A*) (*term-of-gterm* (*map-gterm prod.swap t*))
⟨*proof*⟩

lemma *to-ta-der-pair-automaton*:
p |∈| *ta-der A* (*term-of-gterm t*) ⟹
(*Some p*, *None*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*map-gterm* (λ*f*. (*Some f*, *None*)) *t*))
q |∈| *ta-der B* (*term-of-gterm u*) ⟹
(*None*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*map-gterm* (λ*f*. (*None*, *Some f*)) *u*))
p |∈| *ta-der A* (*term-of-gterm t*) ⟹ *q* |∈| *ta-der B* (*term-of-gterm u*) ⟹
(*Some p*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm* (*gpair t u*))
⟨*proof*⟩

lemma *from-ta-der-pair-automaton*:
(*None*, *None*) |∉| *ta-der* (*pair-automaton A B*) (*term-of-gterm s*)
(*Some p*, *None*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm s*) ⟹
∃ *t*. *p* |∈| *ta-der A* (*term-of-gterm t*) ∧ *s* = *map-gterm* (λ*f*. (*Some f*, *None*)) *t*
(*None*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm s*) ⟹
∃ *u*. *q* |∈| *ta-der B* (*term-of-gterm u*) ∧ *s* = *map-gterm* (λ*f*. (*None*, *Some f*)) *u*
(*Some p*, *Some q*) |∈| *ta-der* (*pair-automaton A B*) (*term-of-gterm s*) ⟹
∃ *t u*. *p* |∈| *ta-der A* (*term-of-gterm t*) ∧ *q* |∈| *ta-der B* (*term-of-gterm u*) ∧ *s* = *gpair t u*
⟨*proof*⟩

lemma *diagonal-automaton*:
assumes *RR1-spec A R*
shows *RR2-spec* (*fmap-funs-reg* (λ*f*. (*Some f*, *Some f*)) *A*) {(*s*, *s*) | *s*. *s* ∈ *R*}
⟨*proof*⟩

lemma *pair-automaton*:
assumes *RR1-spec A T RR1-spec B U*
shows *RR2-spec* (*pair-automaton-reg A B*) (*T* × *U*)
⟨*proof*⟩

lemma *pair-automaton'*:
shows ℒ (*pair-automaton-reg A B*) = *case-prod gpair* ' (ℒ *A* × ℒ *B*)
⟨*proof*⟩

5.8 Collapsing

cf. *gcollapse*.

fun *collapse-state-list* **where**
collapse-state-list Qn Qs [] = [[]]
| *collapse-state-list Qn Qs* (*q* # *qs*) = (*let rec* = *collapse-state-list Qn Qs qs in*
(*if q* |∈| *Qn* ∧ *q* |∈| *Qs then map* (*Cons None*) *rec* @ *map* (*Cons* (*Some q*)) *rec*
else if q |∈| *Qn then map* (*Cons None*) *rec*
else if q |∈| *Qs then map* (*Cons* (*Some q*)) *rec*
else [[]]))

lemma *collapse-state-list-inner-length*:
assumes *qss* = *collapse-state-list Qn Qs qs*
and ∀ *i* < *length qs*. *qs* ! *i* |∈| *Qn* ∨ *qs* ! *i* |∈| *Qs*
and *i* < *length qss*
shows *length* (*qss* ! *i*) = *length qs* ⟨*proof*⟩

lemma *collapse-fset-inv-constr*:
assumes ∀ *i* < *length qs'*. *qs* ! *i* |∈| *Qn* ∧ *qs'* ! *i* = *None* ∨
qs ! *i* |∈| *Qs* ∧ *qs'* ! *i* = *Some* (*qs* ! *i*)
and *length qs* = *length qs'*
shows *qs'* |∈| *fset-of-list* (*collapse-state-list Qn Qs qs*) ⟨*proof*⟩

lemma *collapse-fset-inv-constr2*:
assumes ∀ *i* < *length qs*. *qs* ! *i* |∈| *Qn* ∨ *qs* ! *i* |∈| *Qs*
and *qs'* |∈| *fset-of-list* (*collapse-state-list Qn Qs qs*) **and** *i* < *length qs'*
shows *qs* ! *i* |∈| *Qn* ∧ *qs'* ! *i* = *None* ∨ *qs* ! *i* |∈| *Qs* ∧ *qs'* ! *i* = *Some* (*qs* ! *i*)
⟨*proof*⟩

definition *collapse-rule* **where**
collapse-rule A Qn Qs =
|⋃| ((λ *r*. *fset-of-list* (*map* (λ *qs*. *TA-rule* (*r-root r*) *qs* (*Some* (*r-rhs r*)))
(*collapse-state-list Qn Qs* (*r-lhs-states r*)))) |'|
ffilter (λ *r*. (∀ *i* < *length* (*r-lhs-states r*). *r-lhs-states r* ! *i* |∈| *Qn* ∨ *r-lhs-states*

r ! *i* |∈| *Qs*))
(*ffilter* (λ *r*. *r-root r* ≠ *None*) (*rules A*)))

definition *collapse-rule-fset* **where**
collapse-rule-fset A Qn Qs = (λ *r*. *TA-rule* (*the* (*r-root r*)) (*map the* (*filter* (λ*q*. ¬ *Option.is-none q*) (*r-lhs-states r*))) (*the* (*r-rhs r*))) |'|
collapse-rule A Qn Qs

lemma *collapse-rule-set-conv*:
fset (*collapse-rule-fset A Qn Qs*) = {*TA-rule f* (*map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs'*)) *q* | *f qs qs' q*.
TA-rule (*Some f*) *qs q* |∈| *rules A* ∧ *length qs* = *length qs'* ∧
(∀ *i* < *length qs*. *qs* ! *i* |∈| *Qn* ∧ *qs'* ! *i* = *None* ∨ *qs* ! *i* |∈| *Qs* ∧ (*qs'* ! *i*) = *Some* (*qs* ! *i*))} (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *collapse-rule-fmember* [*simp*]:
TA-rule f qs q |∈| (*collapse-rule-fset A Qn Qs*) ⟷ (∃ *qs' ps*.
qs = *map the* (*filter* (λ*q*. ¬ *Option.is-none q*) *qs'*) ∧ *TA-rule* (*Some f*) *ps q* |∈| *rules A* ∧ *length ps* = *length qs'* ∧
(∀ *i* < *length ps*. *ps* ! *i* |∈| *Qn* ∧ *qs'* ! *i* = *None* ∨ *ps* ! *i* |∈| *Qs* ∧ (*qs'* ! *i*) = *Some* (*ps* ! *i*)))
⟨*proof*⟩

definition *Qn A* ≡ (*let S* = (*r-rhs* |'| *ffilter* (λ *r*. *r-root r* = *None*) (*rules A*)) *in* (*eps A*)$|^+|$ |''| *S* |∪| *S*)
definition *Qs A* ≡ (*let S* = (*r-rhs* |'| *ffilter* (λ *r*. *r-root r* ≠ *None*) (*rules A*)) *in* (*eps A*)$|^+|$ |''| *S* |∪| *S*)

lemma *Qn-member-iff* [*simp*]:
q |∈| *Qn A* ⟷ (∃ *ps p*. *TA-rule None ps p* |∈| *rules A* ∧ (*p* = *q* ∨ (*p*, *q*) |∈| (*eps A*)$|^+|$)) (**is** *?Ls* ⟷ *?Rs*)
⟨*proof*⟩

lemma *Qs-member-iff* [*simp*]:
q |∈| *Qs A* ⟷ (∃ *f ps p*. *TA-rule* (*Some f*) *ps p* |∈| *rules A* ∧ (*p* = *q* ∨ (*p*, *q*) |∈| (*eps A*)$|^+|$)) (**is** *?Ls* ⟷ *?Rs*)
⟨*proof*⟩

lemma *collapse-Qn-Qs-set-conv*:
fset (*Qn A*) = {*q'* |*qs q q'*. *TA-rule None qs q* |∈| *rules A* ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps A*)$|^+|$)} (**is** *?Ls1* = *?Rs1*)
fset (*Qs A*) = {*q'* |*f qs q q'*. *TA-rule* (*Some f*) *qs q* |∈| *rules A* ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps A*)$|^+|$)} (**is** *?Ls2* = *?Rs2*)
⟨*proof*⟩

definition *collapse-automaton* :: ('*q*, '*f option*) *ta* ⇒ ('*q*, '*f*) *ta* **where**

collapse-automaton A = *TA* (*collapse-rule-fset A* (*Qn A*) (*Qs A*)) (*eps A*)

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

lemma *ta-states-collapse-automaton*:
$\mathcal{Q}$ (*collapse-automaton A*) |⊆| $\mathcal{Q}$ *A*
⟨*proof*⟩

lemma *last-nthI*:
assumes *i* < *length ts* ¬ *i* < *length ts* − *Suc 0*
shows *ts* ! *i* = *last ts* ⟨*proof*⟩

lemma *collapse-automaton′*:
assumes $\mathcal{Q}$ *A* |⊆| *ta-reachable A*
shows *gta-lang Q* (*collapse-automaton A*) = *the* ‘ (*gcollapse* ‘ *gta-lang Q A* − {*None*})
⟨*proof*⟩

lemma $\mathcal{L}$*-collapse-automaton′*:
assumes $\mathcal{Q}_r$ *A* |⊆| *ta-reachable* (*ta A*)
shows $\mathcal{L}$ (*collapse-automaton-reg A*) = *the* ‘ (*gcollapse* ‘ $\mathcal{L}$ *A* − {*None*})
⟨*proof*⟩

lemma *collapse-automaton*:
assumes $\mathcal{Q}_r$ *A* |⊆| *ta-reachable* (*ta A*) *RR1-spec A T*
shows *RR1-spec* (*collapse-automaton-reg A*) (*the* ‘ (*gcollapse* ‘ $\mathcal{L}$ *A* − {*None*}))
⟨*proof*⟩

5.9 Cylindrification

definition *pad-with-Nones* **where**
pad-with-Nones n m = (λ(*f*, *g*). *case-option* (*replicate n None*) *id f* @ *case-option* (*replicate m None*) *id g*)

lemma *gencode-append*:
gencode (*ss* @ *ts*) = *map-gterm* (*pad-with-Nones* (*length ss*) (*length ts*)) (*gpair* (*gencode ss*) (*gencode ts*))
⟨*proof*⟩

lemma *append-automaton*:
assumes *RRn-spec n A T RRn-spec m B U*
shows *RRn-spec* (*n* + *m*) (*fmap-funs-reg* (*pad-with-Nones n m*) (*pair-automaton-reg A B*)) {*ts* @ *us* |*ts us*. *ts* ∈ *T* ∧ *us* ∈ *U*}
⟨*proof*⟩

lemma *cons-automaton*:
assumes *RR1-spec A T RRn-spec m B U*
shows *RRn-spec* (*Suc m*) (*fmap-funs-reg* (λ(*f*, *g*). *pad-with-Nones 1 m* (*map-option*

$(\lambda f.\ [Some\ f])\ f,\ g))$
  $(pair\text{-}automaton\text{-}reg\ A\ B))\ \{t\ \#\ us\ |t\ us.\ t \in T \wedge us \in U\}$
⟨*proof*⟩

5.10 Projection

abbreviation *drop-none-rule m fs* ≡ *if list-all* (*Option.is-none*) (*drop m fs*) *then None else Some* (*drop m fs*)

lemma *drop-automaton-reg*:
  assumes $\mathcal{Q}_r$ *A* |⊆| *ta-reachable* (*ta A*) *m* < *n RRn-spec n A T*
  defines *f* ≡ λ*fs. drop-none-rule m fs*
  shows *RRn-spec* (*n* − *m*) (*collapse-automaton-reg* (*fmap-funs-reg f A*)) (*drop m* ‘ *T*)
⟨*proof*⟩

lemma *gfst-collapse-simp*:
  the (*gcollapse* (*map-gterm fst t*)) = *gfst t*
  ⟨*proof*⟩

lemma *gsnd-collapse-simp*:
  the (*gcollapse* (*map-gterm snd t*)) = *gsnd t*
  ⟨*proof*⟩

definition *proj-1-reg* **where**
  proj-1-reg A = *collapse-automaton-reg* (*fmap-funs-reg fst* (*trim-reg A*))
definition *proj-2-reg* **where**
  proj-2-reg A = *collapse-automaton-reg* (*fmap-funs-reg snd* (*trim-reg A*))

lemmas *proj-1-reg-simp* = *proj-1-reg-def collapse-automaton-reg-def fmap-funs-reg-def trim-reg-def*
lemmas *proj-2-reg-simp* = *proj-2-reg-def collapse-automaton-reg-def fmap-funs-reg-def trim-reg-def*

lemma $\mathcal{L}$*-proj-1-reg-collapse*:
  $\mathcal{L}$ (*proj-1-reg* $\mathcal{A}$) = *the* ‘ (*gcollapse* ‘ *map-gterm fst* ‘ ($\mathcal{L}$ $\mathcal{A}$) − {*None*})
⟨*proof*⟩

lemma $\mathcal{L}$*-proj-2-reg-collapse*:
  $\mathcal{L}$ (*proj-2-reg* $\mathcal{A}$) = *the* ‘ (*gcollapse* ‘ *map-gterm snd* ‘ ($\mathcal{L}$ $\mathcal{A}$) − {*None*})
⟨*proof*⟩

lemma *proj-1*:
  assumes *RR2-spec A R*
  shows *RR1-spec* (*proj-1-reg A*) (*fst* ‘ *R*)
⟨*proof*⟩

lemma *proj-2*:
  assumes *RR2-spec A R*

shows *RR1-spec* (*proj-2-reg A*) (*snd* ‘ *R*)
⟨*proof*⟩

lemma *ℒ-proj*:
assumes *RR2-spec A R*
shows ℒ (*proj-1-reg A*) = *gfst* ‘ ℒ *A* ℒ (*proj-2-reg A*) = *gsnd* ‘ ℒ *A*
⟨*proof*⟩

lemmas *proj-automaton-gta-lang* = *proj-1 proj-2*

5.11 Permutation

lemma *gencode-permute*:
assumes *set ps* = {*0..<length ts*}
shows *gencode* (*map* ((!) *ts*) *ps*) = *map-gterm* (λ*xs. map* ((!) *xs*) *ps*) (*gencode ts*)
⟨*proof*⟩

lemma *permute-automaton*:
assumes *RRn-spec n A T set ps* = {*0..<n*}
shows *RRn-spec* (*length ps*) (*fmap-funs-reg* (λ*xs. map* ((!) *xs*) *ps*) *A*) ((λ*xs. map* ((!) *xs*) *ps*) ‘ *T*)
⟨*proof*⟩

5.12 Intersection

lemma *intersect-automaton*:
assumes *RRn-spec n A T RRn-spec n B U*
shows *RRn-spec n* (*reg-intersect A B*) (*T* ∩ *U*) ⟨*proof*⟩

lemma *union-automaton*:
assumes *RRn-spec n A T RRn-spec n B U*
shows *RRn-spec n* (*reg-union A B*) (*T* ∪ *U*)
⟨*proof*⟩

5.13 Difference

lemma *RR1-difference*:
assumes *RR1-spec A T RR1-spec B U*
shows *RR1-spec* (*difference-reg A B*) (*T* − *U*)
⟨*proof*⟩

lemma *RR2-difference*:
assumes *RR2-spec A T RR2-spec B U*
shows *RR2-spec* (*difference-reg A B*) (*T* − *U*)
⟨*proof*⟩

lemma *RRn-difference*:
assumes *RRn-spec n A T RRn-spec n B U*

shows *RRn-spec n* (*difference-reg A B*) ($T - U$)
⟨*proof*⟩

5.14 All terms over a signature

definition *term-automaton* :: ($'f \times nat$) *fset* ⇒ (*unit*, $'f$) *ta* **where**
term-automaton $\mathcal{F}$ = *TA* ((λ (f, n). *TA-rule f* (*replicate n* ())) ()) |'| $\mathcal{F}$) {||}
definition *term-reg* **where**
term-reg $\mathcal{F}$ = *Reg* {|()|} (*term-automaton* $\mathcal{F}$)

lemma *term-automaton*:
RR1-spec (*term-reg* $\mathcal{F}$) ($\mathcal{T}_G$ (*fset* $\mathcal{F}$))
⟨*proof*⟩

fun *true-RRn* :: ($'f \times nat$) *fset* ⇒ *nat* ⇒ (*nat*, $'f$ *option list*) *reg* **where**
true-RRn $\mathcal{F}$ *0* = *Reg* {|*0*|} (*TA* {|*TA-rule* [] [] *0*|} {||})
| *true-RRn* $\mathcal{F}$ (*Suc 0*) = *relabel-reg* (*fmap-funs-reg* (λf. [*Some f*]) (*term-reg* $\mathcal{F}$))
| *true-RRn* $\mathcal{F}$ (*Suc n*) = *relabel-reg*
(*trim-reg* (*fmap-funs-reg* (*pad-with-Nones 1 n*) (*pair-automaton-reg* (*true-RRn* $\mathcal{F}$ *1*) (*true-RRn* $\mathcal{F}$ *n*))))

lemma *true-RRn-spec*:
RRn-spec n (*true-RRn* $\mathcal{F}$ *n*) {*ts*. *length ts* = n ∧ *set ts* ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$)}
⟨*proof*⟩

5.15 RR2 composition

abbreviation *RR2-to-RRn A* ≡ *fmap-funs-reg* (λ(f, g). [f, g]) A
abbreviation *RRn-to-RR2 A* ≡ *fmap-funs-reg* (λf. (f ! *0*, f ! *1*)) A
definition *rr2-compositon* **where**
rr2-compositon $\mathcal{F}$ *A B* =
(*let* A' = *RR2-to-RRn A in*
let B' = *RR2-to-RRn B in*
let F = *true-RRn* $\mathcal{F}$ *1 in*
let CA = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 2 1*) (*pair-automaton-reg* A' *F*)) *in*
let CB = *trim-reg* (*fmap-funs-reg* (*pad-with-Nones 1 2*) (*pair-automaton-reg F* B')) *in*
let PI = *trim-reg* (*fmap-funs-reg* (λ*xs*. *map* ((!) *xs*) [*1*, *0*, *2*]) (*reg-intersect CA CB*)) *in*
RRn-to-RR2 (*collapse-automaton-reg* (*fmap-funs-reg* (*drop-none-rule 1*) *PI*))
)

lemma *list-length1E*:
assumes *length xs* = *Suc 0* **obtains** x **where** *xs* = [x] ⟨*proof*⟩

lemma *rr2-compositon*:
assumes $\mathcal{R} \subseteq \mathcal{T}_G$ (*fset* $\mathcal{F}$) × $\mathcal{T}_G$ (*fset* $\mathcal{F}$) $\mathfrak{L} \subseteq \mathcal{T}_G$ (*fset* $\mathcal{F}$) × $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
and *RR2-spec A* $\mathcal{R}$ **and** *RR2-spec B* $\mathfrak{L}$
shows *RR2-spec* (*rr2-compositon* $\mathcal{F}$ *A B*) ($\mathcal{R}$ *O* $\mathfrak{L}$)

⟨*proof*⟩

end
theory *RR2-Infinite*
  imports *RRn-Automata Tree-Automata-Pumping*
begin

lemma *map-ta-rule-id* [*simp*]: *map-ta-rule f id r* = (*r-root r*) (*map f* (*r-lhs-states r*)) → (*f* (*r-rhs r*)) **for** *f r*
  ⟨*proof*⟩

lemma *no-upper-bound-infinite*:
  assumes ∀ (*n*::*nat*). ∃ *t* ∈ *S*. *n* < *f t*
  shows *infinite S*
⟨*proof*⟩

lemma *set-constr-finite*:
  assumes *finite F*
  shows *finite* {*h x* | *x*. *x* ∈ *F* ∧ *P x*} ⟨*proof*⟩

lemma *bounded-depth-finite*:
  assumes *fin-F*: *finite* $\mathcal{F}$ **and** ⋃ (*funas-term* ‘ *S*) ⊆ $\mathcal{F}$
    and ∀ *t* ∈ *S*. *depth t* ≤ *n* **and** ∀ *t* ∈ *S*. *ground t*
  shows *finite S* ⟨*proof*⟩

lemma *infinite-imageD*:
  infinite (*f* ‘ *S*) ⟹ *inj-on f S* ⟹ *infinite S*
  ⟨*proof*⟩

lemma *infinite-imageD2*:
  infinite (*f* ‘ *S*) ⟹ *inj f* ⟹ *infinite S*
  ⟨*proof*⟩

lemma *infinite-inj-image-infinite*:
  assumes *infinite S* **and** *inj-on f S*
  shows *infinite* (*f* ‘ *S*)
  ⟨*proof*⟩

lemma *infinte-no-depth-limit*:
  assumes *infinite S* **and** *finite* $\mathcal{F}$
    and ∀ *t* ∈ *S*. *funas-term t* ⊆ $\mathcal{F}$ **and** ∀ *t* ∈ *S*. *ground t*
  shows ∀ (*n*::*nat*). ∃ *t* ∈ *S*. *n* < (*depth t*)
⟨*proof*⟩

lemma *depth-gterm-conv*:

depth (*term-of-gterm* t) = *depth* (*term-of-gterm* t)
⟨*proof*⟩

lemma *funs-term-ctxt* [*simp*]:
funs-term $C\langle s\rangle$ = *funs-ctxt* C ∪ *funs-term* s
⟨*proof*⟩

lemma *pigeonhole-ta-infinit-terms*:
fixes t ::′*f gterm* **and** $\mathcal{A}$ :: (′*q*, ′*f*) *ta*
defines t' ≡ *term-of-gterm* t :: (′*f*, ′*q*) *term*
assumes *fcard* ($\mathcal{Q}$ $\mathcal{A}$) < *depth* t' **and** q |∈| *gta-der* $\mathcal{A}$ t **and** P (*funas-gterm* t)
shows *infinite* {t . q |∈| *gta-der* $\mathcal{A}$ t ∧ P (*funas-gterm* t)}
⟨*proof*⟩

lemma *gterm-to-None-Some-funas* [*simp*]:
funas-gterm (*gterm-to-None-Some* t) ⊆ (λ (f, n). ((*None*, *Some* f), n)) ‘ $\mathcal{F}$ ⟷ *funas-gterm* t ⊆ $\mathcal{F}$
⟨*proof*⟩

lemma *funas-gterm-bot-some-decomp*:
assumes *funas-gterm* s ⊆ (λ (f, n). ((*None*, *Some* f), n)) ‘ $\mathcal{F}$
shows ∃ t. *gterm-to-None-Some* t = s ∧ *funas-gterm* t ⊆ $\mathcal{F}$ ⟨*proof*⟩

definition *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$ = {t . *infinite* {u. (t, u) ∈ $\mathcal{R}$ ∧ *funas-gterm* u ⊆ *fset* $\mathcal{F}$} ∧ *funas-gterm* t ⊆ *fset* $\mathcal{F}$}

definition *Q-infty* $\mathcal{A}$ $\mathcal{F}$ = {|q | q. *infinite* {t | t. *funas-gterm* t ⊆ *fset* $\mathcal{F}$ ∧ q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some* t))}|}

lemma *Q-infty-fmember*:
q |∈| *Q-infty* $\mathcal{A}$ $\mathcal{F}$ ⟷ *infinite* {t | t. *funas-gterm* t ⊆ *fset* $\mathcal{F}$ ∧ q |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm* (*gterm-to-None-Some* t))}
⟨*proof*⟩

abbreviation *q-inf-dash-intro-rules* **where**
q-inf-dash-intro-rules Q r ≡ *if* (*r-rhs* r) |∈| Q ∧ *fst* (*r-root* r) = *None* *then* {|(*r-root* r) (*map CInl* (*r-lhs-states* r)) → *CInr* (*r-rhs* r)|} *else* {||}

abbreviation *args* :: ′*a list* ⇒ *nat* ⇒ (′*a* + ′*a*) *list* **where**
args ≡ λ *qs* i. *map CInl* (*take* i *qs*) @ *CInr* (*qs* ! i) # *map CInl* (*drop* (*Suc* i) *qs*)

abbreviation *q-inf-dash-closure-rules* :: (′*q*, ′*f*) *ta-rule* ⇒ (′*q* + ′*q*, ′*f*) *ta-rule list* **where**
q-inf-dash-closure-rules r ≡ (*let* (f, *qs*, q) = (*r-root* r, *r-lhs-states* r, *r-rhs* r) *in*
(*map* (λ i. f (*args qs* i) → *CInr* q) [*0* ..< *length qs*]))

definition *Inf-automata* :: ('*q*, '*f option* × '*f option*) *ta* ⇒ '*q fset* ⇒ ('*q* + '*q*, '*f option* × '*f option*) *ta* **where**
Inf-automata $\mathcal{A}$ *Q* = *TA*
((|⋃| (*q-inf-dash-intro-rules Q* |'| *rules* $\mathcal{A}$)) |∪| (|⋃| ((*fset-of-list* ∘ *q-inf-dash-closure-rules*) |'| *rules* $\mathcal{A}$)) |∪|
map-ta-rule CInl id |'| *rules* $\mathcal{A}$) (*map-both Inl* |'| *eps* $\mathcal{A}$ |∪| *map-both CInr* |'| *eps* $\mathcal{A}$)

definition *Inf-reg* **where**
Inf-reg $\mathcal{A}$ *Q* = *Reg* (*CInr* |'| *fin* $\mathcal{A}$) (*Inf-automata* (*ta* $\mathcal{A}$) *Q*)

lemma *Inr-Inl-rel-comp*:
map-both CInr |'| *S* |*O*| *map-both CInl* |'| *S* = {||} ⟨*proof*⟩

lemmas *eps-split* = *ftrancl-Un2-separatorE*[*OF Inr-Inl-rel-comp*]

lemma *Inf-automata-eps-simp* [*simp*]:
shows (*map-both Inl* |'| *eps* $\mathcal{A}$ |∪| *map-both CInr* |'| *eps* $\mathcal{A}$)|⁺| =
(*map-both CInl* |'| *eps* $\mathcal{A}$)|⁺| |∪| (*map-both CInr* |'| *eps* $\mathcal{A}$)|⁺|
⟨*proof*⟩

lemma *map-both-CInl-ftrancl-conv*:
(*map-both CInl* |'| *eps* $\mathcal{A}$)|⁺| = *map-both CInl* |'| (*eps* $\mathcal{A}$)|⁺|
⟨*proof*⟩

lemma *map-both-CInr-ftrancl-conv*:
(*map-both CInr* |'| *eps* $\mathcal{A}$)|⁺| = *map-both CInr* |'| (*eps* $\mathcal{A}$)|⁺|
⟨*proof*⟩

lemmas *map-both-ftrancl-conv* = *map-both-CInl-ftrancl-conv map-both-CInr-ftrancl-conv*

lemma *Inf-automata-Inl-to-eps* [*simp*]:
(*CInl p*, *CInl q*) |∈| (*map-both CInl* |'| *eps* $\mathcal{A}$)|⁺| ⟷ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|
(*CInr p*, *CInr q*) |∈| (*map-both CInr* |'| *eps* $\mathcal{A}$)|⁺| ⟷ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)|⁺|
(*CInl q*, *CInl p*) |∈| (*map-both CInr* |'| *eps* $\mathcal{A}$)|⁺| ⟷ *False*
(*CInr q*, *CInr p*) |∈| (*map-both CInl* |'| *eps* $\mathcal{A}$)|⁺| ⟷ *False*
⟨*proof*⟩

lemma *Inl-eps-Inr*:
(*CInl q*, *CInl p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|⁺| ⟷ (*CInr q*, *CInr p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|⁺|
⟨*proof*⟩

lemma *Inr-rhs-eps-Inr-lhs*:
assumes (*q*, *CInr p*) |∈| (*eps* (*Inf-automata* $\mathcal{A}$ *Q*))|⁺|
obtains *q'* **where** *q* = *CInr q'* ⟨*proof*⟩

lemma *Inl-rhs-eps-Inl-lhs*:

assumes $(q,\ CInl\ p)\ |{\in}|\ (eps\ (Inf\text{-}automata\ \mathcal{A}\ Q))|^+|$
obtains q' **where** $q = CInl\ q'$ ⟨proof⟩

lemma *Inf-automata-eps* [*simp*]:
$(CInl\ q,\ CInr\ p)\ |{\in}|\ (eps\ (Inf\text{-}automata\ \mathcal{A}\ Q))|^+| \longleftrightarrow False$
$(CInr\ q,\ CInl\ p)\ |{\in}|\ (eps\ (Inf\text{-}automata\ \mathcal{A}\ Q))|^+| \longleftrightarrow False$
⟨proof⟩

lemma *Inl-A-res-Inf-automata*:
$ta\text{-}der\ (fmap\text{-}states\text{-}ta\ CInl\ \mathcal{A})\ t\ |{\subseteq}|\ ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ t$
⟨proof⟩

lemma *Inl-res-A-res-Inf-automata*:
$CInl\ |`|\ ta\text{-}der\ \mathcal{A}\ (term\text{-}of\text{-}gterm\ t)\ |{\subseteq}|\ ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ (term\text{-}of\text{-}gterm\ t)$
⟨proof⟩

lemma *r-rhs-CInl-args-A-rule*:
assumes $f\ qs \rightarrow CInl\ q\ |{\in}|\ rules\ (Inf\text{-}automata\ \mathcal{A}\ Q)$
obtains qs' **where** $qs = map\ CInl\ qs'\ f\ qs' \rightarrow q\ |{\in}|\ rules\ \mathcal{A}$ ⟨proof⟩

lemma *A-rule-to-dash-closure*:
assumes $f\ qs \rightarrow q\ |{\in}|\ rules\ \mathcal{A}$ **and** $i < length\ qs$
shows $f\ (args\ qs\ i) \rightarrow CInr\ q\ |{\in}|\ rules\ (Inf\text{-}automata\ \mathcal{A}\ Q)$
⟨proof⟩

lemma *Inf-automata-reach-to-dash-reach*:
assumes $CInl\ p\ |{\in}|\ ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInl\ q)\rangle$
shows $CInr\ p\ |{\in}|\ ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ C\langle Var\ (CInr\ q)\rangle$ (**is** - $|{\in}|$ *ta-der ?A* -)
⟨proof⟩

lemma *Inf-automata-dashI*:
assumes $run\ \mathcal{A}\ r\ (gterm\text{-}to\text{-}None\text{-}Some\ t)$ **and** $ex\text{-}rule\text{-}state\ r\ |{\in}|\ Q$
shows $CInr\ (ex\text{-}rule\text{-}state\ r)\ |{\in}|\ gta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ (gterm\text{-}to\text{-}None\text{-}Some\ t)$
⟨proof⟩

lemma *Inf-automata-dash-reach-to-reach*:
assumes $p\ |{\in}|\ ta\text{-}der\ (Inf\text{-}automata\ \mathcal{A}\ Q)\ t$ (**is** - $|{\in}|$ *ta-der ?A* -)
shows $remove\text{-}sum\ p\ |{\in}|\ ta\text{-}der\ \mathcal{A}\ (map\text{-}vars\text{-}term\ remove\text{-}sum\ t)$ ⟨proof⟩

lemma *depth-poss-split*:
assumes $Suc\ (depth\ (term\text{-}of\text{-}gterm\ t) + n) < depth\ (term\text{-}of\text{-}gterm\ u)$
shows $\exists\ p\ q.\ p\ @\ q \in gposs\ u \land n < length\ q \land p \notin gposs\ t$
⟨proof⟩

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 ∃ *u. gpair t u* ∈ $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) (**is** ∃ *u. gpair t u* ∈ $\mathcal{L}$ *?B*)
⟨*proof*⟩

lemma *CInr-Inf-automata-to-q-state*:
assumes *CInr p* |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *t* **and** *ground t*
shows ∃ *C s q*. *C*⟨*s*⟩ = *t* ∧ *CInr q* |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *s* ∧ *q* |∈| *Q* ∧
CInr p |∈| *ta-der* (*Inf-automata* $\mathcal{A}$ *Q*) *C*⟨*Var* (*CInr q*)⟩ ∧
(*fst* ∘ *fst* ∘ *the* ∘ *root*) *s* = *None* ⟨*proof*⟩

lemma *aux-lemma*:
assumes *RR2-spec* $\mathcal{A}$ $\mathcal{R}$ **and** $\mathcal{R}$ ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) × $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
and *infinite* {*u* | *u*. *gpair t u* ∈ $\mathcal{L}$ $\mathcal{A}$}
shows *t* ∈ *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
⟨*proof*⟩

lemma *Inf-automata-to-Inf*:
assumes *RR2-spec* $\mathcal{A}$ $\mathcal{R}$ **and** $\mathcal{R}$ ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) × $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
and *gpair t u* ∈ $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$))
shows *t* ∈ *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
⟨*proof*⟩

lemma *Inf-automata-subseteq*:
$\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) ⊆ $\mathcal{L}$ $\mathcal{A}$ (**is** $\mathcal{L}$ *?IA* ⊆ -)
⟨*proof*⟩

lemma $\mathcal{L}$*-Inf-reg*:
assumes *RR2-spec* $\mathcal{A}$ $\mathcal{R}$ **and** $\mathcal{R}$ ⊆ $\mathcal{T}_G$ (*fset* $\mathcal{F}$) × $\mathcal{T}_G$ (*fset* $\mathcal{F}$)
shows *gfst* ‘ $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$)) = *Inf-branching-terms* $\mathcal{R}$ $\mathcal{F}$
⟨*proof*⟩
end
theory *Tree-Automata-Abstract-Impl*
imports *Tree-Automata-Det Horn-Fset*
begin

6 Computing state derivation

lemma *ta-der-Var-code* [*code*]:
ta-der $\mathcal{A}$ (*Var q*) = *finsert q* ((*eps* $\mathcal{A}$)$|^+|$ |‘‘| {|*q*|})
⟨*proof*⟩

lemma *ta-der-Fun-code* [*code*]:
ta-der $\mathcal{A}$ (*Fun f ts*) =
(*let args* = *map* (*ta-der* $\mathcal{A}$) *ts in*
let P = (λ *r*. *case r of TA-rule g ps p* ⇒ *f* = *g* ∧ *list-all2 fmember ps args*) *in*
let S = *r-rhs* |‘| *ffilter P* (*rules* $\mathcal{A}$) *in*
S |∪| (*eps* $\mathcal{A}$)$|^+|$ |‘‘| *S*) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

definition *eps-free-automata* **where**
eps-free-automata epscl $\mathcal{A}$ =
(*let ruleps* = (λ *r*. *finsert* (*r-rhs r*) (*epscl* |‘‘| {|*r-rhs r*|})) *in*
let rules = (λ *r*. (λ *q*. *TA-rule* (*r-root r*) (*r-lhs-states r*) *q*) |‘| (*ruleps r*)) |‘| (*rules* $\mathcal{A}$) *in*
TA (|⋃| *rules*) {||})

lemma *eps-free* [*code*]:
eps-free $\mathcal{A}$ = *eps-free-automata* ((*eps* $\mathcal{A}$)$|^+|$) $\mathcal{A}$
⟨*proof*⟩

lemma *eps-of-eps-free-automata* [*simp*]:
eps (*eps-free-automata S* $\mathcal{A}$) = {||}
⟨*proof*⟩

lemma *eps-free-automata-empty* [*simp*]:
eps $\mathcal{A}$ = {||} ⟹ *eps-free-automata* {||} $\mathcal{A}$ = $\mathcal{A}$
⟨*proof*⟩

7 Computing the restriction of tree automata to state set

lemma *ta-restrict* [*code*]:
ta-restrict $\mathcal{A}$ *Q* =
(*let rules* = *ffilter* (λ *r*. *case r of TA-rule f ps p* ⇒ *fset-of-list ps* |⊆| *Q* ∧ *p* |∈| *Q*) (*rules* $\mathcal{A}$) *in*
let eps = *ffilter* (λ *r*. *case r of* (*p*, *q*) ⇒ *p* |∈| *Q* ∧ *q* |∈| *Q*) (*eps* $\mathcal{A}$) *in*
TA rules eps)
⟨*proof*⟩

8 Computing the epsilon transition for the product automaton

lemma *prod-eps*[*code-unfold*]:
fCollect (*prod-epsLp* $\mathcal{A}$ $\mathcal{B}$) = (λ ((*p*, *q*), *r*). ((*p*, *r*), (*q*, *r*))) |‘| (*eps* $\mathcal{A}$ |×| $\mathcal{Q}$ $\mathcal{B}$)
fCollect (*prod-epsRp* $\mathcal{A}$ $\mathcal{B}$) = (λ ((*p*, *q*), *r*). ((*r*, *p*), (*r*, *q*))) |‘| (*eps* $\mathcal{B}$ |×| $\mathcal{Q}$ $\mathcal{A}$)
⟨*proof*⟩

9 Computing reachability

inductive-set *ta-reach* **for** $\mathcal{A}$ **where**
rule [*intro*]: *f qs* → *q* |∈| *rules* $\mathcal{A}$ ⟹ ∀ *i* < *length qs*. *qs* ! *i* ∈ *ta-reach* $\mathcal{A}$ ⟹ *q* ∈ *ta-reach* $\mathcal{A}$
| *eps* [*intro*]: *q* ∈ *ta-reach* $\mathcal{A}$ ⟹ (*q*, *r*) |∈| *eps* $\mathcal{A}$ ⟹ *r* ∈ *ta-reach* $\mathcal{A}$

lemma *ta-reach-eps-transI*:
  assumes $(p,\ q)\ |\in|\ (eps\ \mathcal{A})|^+|\ p \in ta\text{-}reach\ \mathcal{A}$
  shows $q \in ta\text{-}reach\ \mathcal{A}$ ⟨*proof*⟩

lemma *ta-reach-ground-term-der*:
  assumes $q \in ta\text{-}reach\ \mathcal{A}$
  shows $\exists\ t.\ ground\ t \wedge q\ |\in|\ ta\text{-}der\ \mathcal{A}\ t$ ⟨*proof*⟩

lemma *ground-term-der-ta-reach*:
  assumes $ground\ t\ q\ |\in|\ ta\text{-}der\ \mathcal{A}\ t$
  shows $q \in ta\text{-}reach\ \mathcal{A}$ ⟨*proof*⟩

lemma *ta-reach-reachable*:
  $ta\text{-}reach\ \mathcal{A} = fset\ (ta\text{-}reachable\ \mathcal{A})$
  ⟨*proof*⟩

9.1 Horn setup for reachable states

definition *reach-rules* $\mathcal{A}$ =
  $\{qs \to_h q \mid f\ qs\ q.\ TA\text{-}rule\ f\ qs\ q\ |\in|\ rules\ \mathcal{A}\} \cup$
  $\{[q] \to_h r \mid q\ r.\ (q,\ r)\ |\in|\ eps\ \mathcal{A}\}$

locale *reach-horn* =
  fixes $\mathcal{A} :: ('q,\ 'f)\ ta$
begin

sublocale *horn reach-rules* $\mathcal{A}$ ⟨*proof*⟩

lemma *reach-infer0*: $infer0 = \{q \mid f\ q.\ TA\text{-}rule\ f\ []\ q\ |\in|\ rules\ \mathcal{A}\}$
  ⟨*proof*⟩

lemma *reach-infer1*:
  $infer1\ p\ X = \{r \mid f\ qs\ r.\ TA\text{-}rule\ f\ qs\ r\ |\in|\ rules\ \mathcal{A} \wedge p \in set\ qs \wedge set\ qs \subseteq insert\ p\ X\} \cup$
   $\{r \mid r.\ (p,\ r)\ |\in|\ eps\ \mathcal{A}\}$
  ⟨*proof*⟩

lemma *reach-sound*:
  $ta\text{-}reach\ \mathcal{A} = saturate$
⟨*proof*⟩
end

9.2 Computing productivity

First, use an alternative definition of productivity

inductive-set $ta\text{-}productive\text{-}ind :: 'q\ fset \Rightarrow ('q,'f)\ ta \Rightarrow 'q\ set$ **for** P **and** $\mathcal{A} :: ('q,'f)\ ta$ **where**
  basic [*intro*]: $q\ |\in|\ P \Longrightarrow q \in ta\text{-}productive\text{-}ind\ P\ \mathcal{A}$

| *eps* [*intro*]: (p, q) |∈| $(eps\ \mathcal{A})|^+|$ $\Longrightarrow q \in$ *ta-productive-ind P* $\mathcal{A} \Longrightarrow p \in$ *ta-productive-ind* *P* $\mathcal{A}$
| *rule*: *TA-rule f qs q* |∈| *rules* $\mathcal{A} \Longrightarrow q \in$ *ta-productive-ind P* $\mathcal{A} \Longrightarrow q' \in$ *set qs* $\Longrightarrow q' \in$ *ta-productive-ind P* $\mathcal{A}$

lemma *ta-productive-ind*:
ta-productive-ind P $\mathcal{A}$ = *fset* (*ta-productive P* $\mathcal{A}$) (**is** *?LS* = *?RS*)
⟨*proof*⟩

9.2.1 Horn setup for productive states

definition *productive-rules P* $\mathcal{A}$ = {[] $\rightarrow_h$ *q* | *q*. *q* |∈| *P*} ∪
{[*r*] $\rightarrow_h$ *q* | *q r*. (*q*, *r*) |∈| *eps* $\mathcal{A}$} ∪
{[*q*] $\rightarrow_h$ *r* | *f qs q r*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ *r* ∈ *set qs*}

locale *productive-horn* =
fixes $\mathcal{A}$ :: ('*q*, '*f*) *ta* **and** *P* :: '*q fset*
begin

sublocale *horn productive-rules P* $\mathcal{A}$ ⟨*proof*⟩

lemma *productive-infer0*: *infer0* = *fset P*
⟨*proof*⟩

lemma *productive-infer1*:
infer1 p X = {*r* | *r*. (*r*, *p*) |∈| *eps* $\mathcal{A}$} ∪
{*r* | *f qs r*. *TA-rule f qs p* |∈| *rules* $\mathcal{A}$ ∧ *r* ∈ *set qs*}
⟨*proof*⟩

lemma *productive-sound*:
ta-productive-ind P $\mathcal{A}$ = *saturate*
⟨*proof*⟩
end

9.3 Horn setup for power set construction states

lemma *prod-list-exists*:
assumes *fst p* ∈ *set qs set qs* ⊆ *insert* (*fst p*) (*fst* ‘ *X*)
obtains *as* **where** *p* ∈ *set as map fst as* = *qs set as* ⊆ *insert p X*
⟨*proof*⟩

definition *ps-states-rules* $\mathcal{A}$ = {*rs* $\rightarrow_h$ (*Wrapp q*) | *rs f q*.
q = *ps-reachable-states* $\mathcal{A}$ *f* (*map ex rs*) ∧ *q* ≠ {||}}

locale *ps-states-horn* =
fixes $\mathcal{A}$ :: ('*q*, '*f*) *ta*
begin

sublocale *horn ps-states-rules* $\mathcal{A}$ ⟨*proof*⟩

lemma *ps-construction-infer0*: *infer0* =
{*Wrapp q* | *f q*. *q* = *ps-reachable-states* $\mathcal{A}$ *f* [] ∧ *q* ≠ {||}}
⟨*proof*⟩

lemma *ps-construction-infer1*:
infer1 p X = {*Wrapp q* | *f qs q*. *q* = *ps-reachable-states* $\mathcal{A}$ *f* (*map ex qs*) ∧ *q* ≠ {||} ∧
p ∈ *set qs* ∧ *set qs* ⊆ *insert p X*}
⟨*proof*⟩

lemma *ps-states-sound*:
ps-states-set $\mathcal{A}$ = *saturate*
⟨*proof*⟩

end

definition *ps-reachable-states-cont* **where**
ps-reachable-states-cont Δ Δ_ε *f ps* =
(*let R* = *ffilter* (λ *r*. *case r of TA-rule g qs q* ⇒ *f* = *g* ∧ *list-all2* (|∈|) *qs ps*) Δ *in*
let S = *r-rhs* |'| *R in*
S |∪| Δ_ε|$^+$| |''| *S*)

lemma *ps-reachable-states* [*code*]:
ps-reachable-states (*TA* Δ Δ_ε) *f ps* = *ps-reachable-states-cont* Δ Δ_ε *f ps*
⟨*proof*⟩

definition *ps-rules-cont* **where**
ps-rules-cont $\mathcal{A}$ *Q* =
(*let sig* = *ta-sig* $\mathcal{A}$ *in*
let qss = (λ (*f*, *n*). (*f*, *n*, *fset-of-list* (*List.n-lists n* (*sorted-list-of-fset Q*)))) |'| *sig in*
let res = (λ (*f*, *n*, *Qs*). (λ *qs*. *TA-rule f qs* (*Wrapp* (*ps-reachable-states* $\mathcal{A}$ *f* (*map ex qs*)))) |'| *Qs*) |'| *qss in*
ffilter (λ *r*. *ex* (*r-rhs r*) ≠ {||}) (|⋃| *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

begin

derive *linorder ta-rule*
derive *linorder term*
derive *compare term*
derive (*compare*) *ccompare term*
derive *ceq ta-rule*
derive (*eq*) *ceq fset*
derive (*eq*) *ceq FSet-Lex-Wrapper*
derive (*no*) *cenum ta-rule*
derive (*no*) *cenum FSet-Lex-Wrapper*
derive *ccompare ta-rule*
derive (*eq*) *ceq term ctxt*
derive (*no*) *cenum term*
derive (*rbt*) *set-impl fset FSet-Lex-Wrapper ta-rule term*

instantiation *fset* :: (*linorder*) *compare*
begin
definition *compare-fset* :: ($'a$ *fset* ⇒ $'a$ *fset* ⇒ *order*)
  where *compare-fset* = (λ *A B*.
    (*let* A' = *sorted-list-of-fset A in*
     let B' = *sorted-list-of-fset B in*
     if $A' < B'$ *then Lt else if* $B' < A'$ *then Gt else Eq*))
instance
  ⟨*proof*⟩
end

instantiation *fset* :: (*linorder*) *ccompare*
begin
definition *ccompare-fset* :: ($'a$ *fset* ⇒ $'a$ *fset* ⇒ *order*) *option*
  where *ccompare-fset* = *Some* (λ *A B*.
    (*let* A' = *sorted-list-of-fset A in*
     let B' = *sorted-list-of-fset B in*
     if $A' < B'$ *then Lt else if* $B' < A'$ *then Gt else Eq*))
instance
  ⟨*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*) *ccompare*
begin

definition *ccompare-FSet-Lex-Wrapper* :: (*'a FSet-Lex-Wrapper* ⇒ *'a FSet-Lex-Wrapper* ⇒ *order*) *option*
where *ccompare-FSet-Lex-Wrapper* = *Some* (λ *A B*.
(*let A'* = *sorted-list-of-fset* (*ex A*) *in*
let B' = *sorted-list-of-fset* (*ex B*) *in*
if A' < *B' then Lt else if B'* < *A' then Gt else Eq*))

instance
⟨*proof*⟩
end

lemma *infinite-ta-rule-UNIV*[*simp*, *intro*]: *infinite* (*UNIV* :: (*'q*,*'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* = (λ (- :: (*'a*,*'b*)*ta-rule option*) - . *False*)
instance ⟨*proof*⟩
end

lemma *finite-finite-Fpow*:
assumes *finite A*
shows *finite* (*Fpow A*) ⟨*proof*⟩

lemma *infinite-infinite-Fpow*:
assumes *infinite A*
shows *infinite* (*Fpow A*)
⟨*proof*⟩

lemma *inj-on-Abs-fset*:
(⋀ *X*. *X* ∈ *A* ⟹ *finite X*) ⟹ *inj-on Abs-fset A* ⟨*proof*⟩

lemma *UNIV-FSet-Lex-Wrapper*:
(*UNIV* :: *'a FSet-Lex-Wrapper set*) = (*Wrapp* ∘ *Abs-fset*) ‘ (*Fpow* (*UNIV* :: *'a set*))

⟨*proof*⟩

lemma *FSet-Lex-Wrapper-UNIV*:
(*UNIV* :: ′*a FSet-Lex-Wrapper set*) = (*Wrapp* ∘ *Abs-fset*) ‘ (*Fpow* (*UNIV* :: ′*a set*))
⟨*proof*⟩

lemma *Wrapp-Abs-fset-inj*:
inj-on (*Wrapp* ∘ *Abs-fset*) (*Fpow A*)
⟨*proof*⟩

lemma *infinite-FSet-Lex-Wrapper-UNIV*:
assumes *infinite* (*UNIV* :: ′*a set*)
shows *infinite* (*UNIV* :: ′*a FSet-Lex-Wrapper set*)
⟨*proof*⟩

lemma *finite-FSet-Lex-Wrapper-UNIV*:
assumes *finite* (*UNIV* :: ′*a set*)
shows *finite* (*UNIV* :: ′*a FSet-Lex-Wrapper set*) ⟨*proof*⟩

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 ⟨*proof*⟩
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* (λ*x m*. *Mapping.update* (*ek x*) (*ev x* @ *case-option Nil id* (*Mapping.lookup m* (*ek x*))) *m*) *xs Mapping.empty*

abbreviation *map-of-list ek ev xs* ≡ *map-val-of-list ek* (λ *x*. [*ev x*]) *xs*

lemma *map-val-of-list-tabulate-conv*:
map-val-of-list ek ev xs = *Mapping.tabulate* (*sort* (*remdups* (*map ek xs*))) (λ *k*. *concat* (*map ev* (*filter* (λ *x*. *k* = *ek x*) *xs*)))
⟨*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_ε =
(*let rules* = *sorted-list-of-fset* Δ *in*
let eps = *sorted-list-of-fset* Δ_ε *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_ε :: (*'q* × *'q*) *fset*
begin

sublocale *reach-horn TA* Δ Δ_ε ⟨*proof*⟩

lemma *infer1*:
infer1 p (*fset bs*) = *set* (*reach-infer1-cont* Δ Δ_ε *p bs*)
⟨*proof*⟩

sublocale *l*: *horn-fset reach-rules* (*TA* Δ Δ_ε) *reach-infer0-cont* Δ *reach-infer1-cont* Δ Δ_ε
⟨*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* Δ Δ_ε =
horn-fset-impl.saturate-impl (*reach-infer0-cont* Δ) (*reach-infer1-cont* Δ Δ_ε)

lemma *reach-fset-impl-sound*:
reach-cont-impl Δ Δ_ε = *Some xs* ⟹ *fset xs* = *ta-reach* (*TA* Δ Δ_ε)
⟨*proof*⟩

lemma *reach-fset-impl-complete*:
reach-cont-impl Δ Δ_ε ≠ *None*
⟨*proof*⟩

lemma *reach-impl* [*code*]:
ta-reachable (*TA* Δ Δ_ε) = *the* (*reach-cont-impl* Δ Δ_ε)

⟨*proof*⟩

9.5 Setup for list implementation of productive states

definition *productive-infer1-cont* :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* ⇒ (*'q* × *'q*) *fset* ⇒ *'q* ⇒ *'q fset* ⇒ *'q list* **where**
productive-infer1-cont Δ Δ_ε =
(*let rules* = *sorted-list-of-fset* Δ *in*
let eps = *sorted-list-of-fset* Δ_ε *in*
let mapp-r = *map-of-list* (λ *r*. *r-rhs r*) *r-lhs-states rules in*
let mapp-e = *map-of-list snd fst eps in*
(λ *p bs*.
(*case-option Nil id* (*Mapping.lookup mapp-e p*)) @
concat (*case-option Nil id* (*Mapping.lookup mapp-r p*))))

locale *productive-rules-fset* =
fixes Δ :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* **and** Δ_ε :: (*'q* × *'q*) *fset* **and** *P* :: *'q fset*
begin

sublocale *productive-horn TA* Δ Δ_ε *P* ⟨*proof*⟩

lemma *infer1*:
infer1 p (*fset bs*) = *set* (*productive-infer1-cont* Δ Δ_ε *p bs*)
⟨*proof*⟩

sublocale *l*: *horn-fset productive-rules P* (*TA* Δ Δ_ε) *sorted-list-of-fset P productive-infer1-cont* Δ Δ_ε
⟨*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* Δ Δ_ε =
horn-fset-impl.saturate-impl (*sorted-list-of-fset P*) (*productive-infer1-cont* Δ Δ_ε)

lemma *productive-cont-impl-sound*:
productive-cont-impl P Δ Δ_ε = *Some xs* ⟹ *fset xs* = *ta-productive-ind P* (*TA* Δ Δ_ε)
⟨*proof*⟩

lemma *productive-cont-impl-complete*:
productive-cont-impl P Δ Δ_ε ≠ *None*
⟨*proof*⟩

lemma *productive-impl* [*code*]:

ta-productive P (*TA* Δ Δ_ε) = *the* (*productive-cont-impl P* Δ Δ_ε)
⟨*proof*⟩

9.6 Setup for the implementation of power set construction states

abbreviation *r-statesl r* ≡ *length* (*r-lhs-states r*)

definition *ps-reachable-states-list* **where**
ps-reachable-states-list mapp-r mapp-e f ps =
(*let R* = *filter* (λ *r*. *list-all2* (|∈|) (*r-lhs-states r*) *ps*)
(*case-option Nil id* (*Mapping.lookup mapp-r* (*f*, *length ps*))) *in*
let S = *map r-rhs R in*
S @ *concat* (*map* (*case-option Nil id* ∘ *Mapping.lookup mapp-e*) *S*))

lemma *ps-reachable-states-list-sound*:
assumes *length ps* = *n*
and *mapp-r*: *case-option Nil id* (*Mapping.lookup mapp-r* (*f*, *n*)) =
filter (λ*r*. *r-root r* = *f* ∧ *r-statesl r* = *n*) (*sorted-list-of-fset* Δ)
and *mapp-e*: ⋀*p*. *case-option Nil id* (*Mapping.lookup mapp-e p*) =
map snd (*filter* (λ *q*. *fst q* = *p*) (*sorted-list-of-fset* (Δ_ε|$^+$|)))
shows *fset-of-list* (*ps-reachable-states-list mapp-r mapp-e f* (*map ex ps*)) =
ps-reachable-states (*TA* Δ Δ_ε) *f* (*map ex ps*) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

lemma *rule-target-statesI*:
∃ *r* |∈| Δ. *r-rhs r* = *q* ⟹ *q* |∈| *rule-target-states* Δ
⟨*proof*⟩

definition *ps-states-infer0-cont* :: (′*q* :: *linorder*, ′*f* :: *linorder*) *ta-rule fset* ⇒
(′*q* × ′*q*) *fset* ⇒ ′*q FSet-Lex-Wrapper list* **where**
ps-states-infer0-cont Δ Δ_ε =
(*let sig* = *filter* (λ *r*. *r-lhs-states r* = []) (*sorted-list-of-fset* Δ) *in*
filter (λ *p*. *ex p* ≠ {||}) (*map* (λ *r*. *Wrapp* (*ps-reachable-states* (*TA* Δ Δ_ε)
(*r-root r*) [])) *sig*))

definition *ps-states-infer1-cont* :: (′*q* :: *linorder* , ′*f* :: *linorder*) *ta-rule fset* ⇒ (′*q*
× ′*q*) *fset* ⇒
′*q FSet-Lex-Wrapper* ⇒ ′*q FSet-Lex-Wrapper fset* ⇒ ′*q FSet-Lex-Wrapper list*
where
ps-states-infer1-cont Δ Δ_ε =
(*let sig* = *remdups* (*map* (λ *r*. (*r-root r*, *r-statesl r*)) (*filter* (λ *r*. *r-lhs-states r*
≠ []) (*sorted-list-of-fset* Δ))) *in*
let arities = *remdups* (*map snd sig*) *in*
let etr = *sorted-list-of-fset* (Δ_ε|$^+$|) *in*
let mapp-r = *map-of-list* (λ *r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* Δ)
in
let mapp-e = *map-of-list fst snd etr in*

(λ *p bs.*
(*let states* = *sorted-list-of-fset* (*finsert p bs*) *in*
let arity-to-states-map = *Mapping.tabulate arities* (λ *n. list-of-permutation-element-n p n states*) *in*
let res = *map* (λ (*f*, *n*).
map (λ *s. let rules* = *the* (*Mapping.lookup mapp-r* (*f*, *n*)) *in*
Wrapp (*fset-of-list* (*ps-reachable-states-list mapp-r mapp-e f* (*map ex s*))))
(*the* (*Mapping.lookup arity-to-states-map n*)))
sig in
filter (λ *p. ex p* $\neq$ {||}) (*concat res*))))

locale *ps-states-fset* =
fixes Δ :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* **and** Δ_ε :: (*'q* $\times$ *'q*) *fset*
begin

sublocale *ps-states-horn TA* Δ Δ_ε ⟨*proof*⟩

lemma *infer0*: *infer0* = *set* (*ps-states-infer0-cont* Δ Δ_ε)
⟨*proof*⟩

lemma *r-lhs-states-nConst*:
r-lhs-states r $\neq$ [] $\Longrightarrow$ *r-statesl r* $\neq$ *0* **for** *r* ⟨*proof*⟩

lemma *filter-empty-conv'*:
[] = *filter P xs* $\longleftrightarrow$ ($\forall$ *x*$\in$*set xs.* $\neg$ *P x*)
⟨*proof*⟩

lemma *infer1*:
infer1 p (*fset bs*) = *set* (*ps-states-infer1-cont* Δ Δ_ε *p bs*) (**is** *?Ls* = *?Rs*)
⟨*proof*⟩

sublocale *l*: *horn-fset ps-states-rules* (*TA* Δ Δ_ε) *ps-states-infer0-cont* Δ Δ_ε *ps-states-infer1-cont* Δ Δ_ε
⟨*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* Δ Δ_ε =
horn-fset-impl.saturate-impl (*ps-states-infer0-cont* Δ Δ_ε) (*ps-states-infer1-cont* Δ Δ_ε)

lemma *ps-states-fset-impl-sound*:

assumes *ps-states-fset-impl* Δ Δ_ε = *Some xs*
shows *xs* = *ps-states* (*TA* Δ Δ_ε)
⟨*proof*⟩

lemma *ps-states-fset-impl-complete*:
ps-states-fset-impl Δ Δ_ε ≠ *None*
⟨*proof*⟩

lemma *ps-ta-impl* [*code*]:
ps-ta (*TA* Δ Δ_ε) =
(*let xs* = *the* (*ps-states-fset-impl* Δ Δ_ε) *in*
TA (*ps-rules* (*TA* Δ Δ_ε) *xs*) {||})
⟨*proof*⟩

lemma *ps-reg-impl* [*code*]:
ps-reg (*Reg Q* (*TA* Δ Δ_ε)) =
(*let xs* = *the* (*ps-states-fset-impl* Δ Δ_ε) *in*
Reg (*ffilter* (λ *S*. *Q* |∩| *ex S* ≠ {||}) *xs*)
(*TA* (*ps-rules* (*TA* Δ Δ_ε) *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* $\mathcal{A}$ |∩| *ta-sig* $\mathcal{B}$) *in*
let mapA = *map-of-list* (λ*r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* (*rules* $\mathcal{A}$)) *in*
let mapB = *map-of-list* (λ*r*. (*r-root r*, *r-statesl r*)) *id* (*sorted-list-of-fset* (*rules* $\mathcal{B}$)) *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*′:
$b = c \Longrightarrow x = u \Longrightarrow y = v \Longrightarrow$ (*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 $\mathcal{A}$ (*Var q*) = {|*q*|}
| *ta-der-strict* $\mathcal{A}$ (*Fun f ts*) = {| *q'* | *q' q qs*. *TA-rule f qs q* |∈| *rules* $\mathcal{A}$ ∧ (*q* = *q'* ∨ (*q*, *q'*) |∈| (*eps* $\mathcal{A}$)$|^+|$) ∧
length qs = *length ts* ∧ (∀ *i* < *length ts*. *qs* ! *i* |∈| *ta-der-strict* $\mathcal{A}$ (*ts* ! *i*))|}

lemma *ta-der-strict-Var*:
q |∈| *ta-der-strict* $\mathcal{A}$ (*Var x*) ⟷ *x* = *q*
⟨*proof*⟩

lemma *ta-der-strict-Fun*:
q |∈| *ta-der-strict* $\mathcal{A}$ (*Fun f ts*) ⟷ (∃ *ps p*. *TA-rule f ps p* |∈| (*rules* $\mathcal{A}$) ∧
(*p* = *q* ∨ (*p*, *q*) |∈| (*eps* $\mathcal{A}$)$|^+|$) ∧ *length ps* = *length ts* ∧
(∀ *i* < *length ts*. *ps* ! *i* |∈| *ta-der-strict* $\mathcal{A}$ (*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 $\mathcal{A}$ *t* |⊆| *ta-der* $\mathcal{A}$ *t*
⟨*proof*⟩

lemma *ta-der-strict-ta-der-eq-on-ground*:
assumes*ground t*
shows *ta-der* $\mathcal{A}$ *t* = *ta-der-strict* $\mathcal{A}$ *t*
⟨*proof*⟩

lemma *ta-der-to-ta-strict*:
assumes *q* |∈| *ta-der A C*⟨*Var p*⟩ **and** *ground-ctxt C*
shows ∃ *q'*. (*p* = *q'* ∨ (*p*, *q'*) |∈| (*eps A*)$|^+|$) ∧ *q* |∈| *ta-der-strict A C*⟨*Var q'*⟩
⟨*proof*⟩

fun *root-ctxt* **where**
root-ctxt (*More f ss C ts*) = *f*
| *root-ctxt* □ = *undefined*

lemma *root-to-root-ctxt* [*simp*]:
assumes *C* ≠ □
shows *fst* (*the* (*root C*⟨*t*⟩)) ⟷ *root-ctxt C*
⟨*proof*⟩

inductive-set *Q-inf* **for** $\mathcal{A}$ **where**

trans: $(p, q) \in$ *Q-inf* $\mathcal{A} \Longrightarrow (q, r) \in$ *Q-inf* $\mathcal{A} \Longrightarrow (p, r) \in$ *Q-inf* $\mathcal{A}$
| *rule*: (*None*, *Some f*) *qs* $\rightarrow$ *q* $|\in|$ *rules* $\mathcal{A} \Longrightarrow i <$ *length qs* $\Longrightarrow$ (*qs* ! *i*, *q*) $\in$ *Q-inf* $\mathcal{A}$
| *eps*: $(p, q) \in$ *Q-inf* $\mathcal{A} \Longrightarrow (q, r)$ $|\in|$ *eps* $\mathcal{A} \Longrightarrow (p, r) \in$ *Q-inf* $\mathcal{A}$

abbreviation *Q-inf-e* $\mathcal{A} \equiv \{q \mid p\ q.\ (p, p) \in$ *Q-inf* $\mathcal{A} \wedge (p, q) \in$ *Q-inf* $\mathcal{A}\}$

lemma *Q-inf-states-ta-states*:
assumes $(p, q) \in$ *Q-inf* $\mathcal{A}$
shows *p* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$ *q* $|\in|$ $\mathcal{Q}$ $\mathcal{A}$
⟨*proof*⟩

lemma *Q-inf-finite*:
finite (*Q-inf* $\mathcal{A}$) *finite* (*Q-inf-e* $\mathcal{A}$)
⟨*proof*⟩

context
includes *fset.lifting*
begin
lift-definition *fQ-inf* :: (*'a*, *'b option* × *'c option*) *ta* ⇒ (*'a* × *'a*) *fset* **is** *Q-inf*
⟨*proof*⟩
lift-definition *fQ-inf-e* :: (*'a*, *'b option* × *'c option*) *ta* ⇒ *'a fset* **is** *Q-inf-e*
⟨*proof*⟩
end

lemma *Q-inf-ta-eps-Q-inf*:
assumes $(p, q) \in$ *Q-inf* $\mathcal{A}$ **and** (q, q') $|\in|$ (*eps* $\mathcal{A})|^+|$
shows $(p, q') \in$ *Q-inf* $\mathcal{A}$ ⟨*proof*⟩

lemma *lhs-state-rule*:
assumes $(p, q) \in$ *Q-inf* $\mathcal{A}$
shows $\exists$ *f qs r*. (*None*, *Some f*) *qs* $\rightarrow$ *r* $|\in|$ *rules* $\mathcal{A} \wedge$ *p* $|\in|$ *fset-of-list qs*
⟨*proof*⟩

lemma *Q-inf-reach-state-rule*:
assumes $(p, q) \in$ *Q-inf* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* $\mathcal{A}$
shows $\exists$ *ss ts f C*. *q* $|\in|$ *ta-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*) *qs* $\rightarrow$ *q'* $|\in|$ *rules* $\mathcal{A}$ **and** $i <$ *length qs*
shows (*qs* ! *i*, *q'*) $\in$ *Q-inf* $\mathcal{A}$ ⟨*proof*⟩

lemma *rule-target-eps-Q-inf*:
assumes (*None*, *Some f*) *qs* $\rightarrow$ *q'* $|\in|$ *rules* $\mathcal{A}$ (q', q) $|\in|$ (*eps* $\mathcal{A})|^+|$
and $i <$ *length qs*

shows $(qs\ !\ i,\ q) \in$ *Q-inf* $\mathcal{A}$
⟨*proof*⟩

lemma *step-in-Q-inf*:
assumes q |∈| *ta-der-strict* $\mathcal{A}$ (*map-funs-term* (λ*f*. (*None*, *Some f*)) (*Fun f* (*ss* @ *Var p* # *ts*)))
shows $(p,\ q) \in$ *Q-inf* $\mathcal{A}$
⟨*proof*⟩

lemma *ta-der-Q-inf*:
assumes q |∈| *ta-der-strict* $\mathcal{A}$ (*map-funs-term* (λ*f*. (*None*, *Some f*)) (*C*⟨*Var p*⟩))
and $C \neq$ *Hole*
shows $(p,\ q) \in$ *Q-inf* $\mathcal{A}$ ⟨*proof*⟩

lemma *Q-inf-e-infinite-terms-res*:
assumes $q \in$ *Q-inf-e* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$
shows *infinite* {*t*. *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) ∧ *fst* (*groot-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 = *gfun-at* t ($p -_p$ *ghole-pos* C) ⟨*proof*⟩

lemma *pos-diff-0* [*simp*]: $p -_p p = []$
⟨*proof*⟩

lemma *Max-suffI*: *finite* $A \Longrightarrow A = B \Longrightarrow$ *Max* A = *Max* B
⟨*proof*⟩

lemma *nth-args-depth-eqI*:
assumes *length ss* = *length ts*
and ⋀ *i*. *i* < *length ts* ⟹ *depth* (*ss* ! *i*) = *depth* (*ts* ! *i*)
shows *depth* (*Fun f ss*) = *depth* (*Fun g ts*)
⟨*proof*⟩

lemma *subt-at-ctxt-apply-hole-pos* [*simp*]: $C\langle s\rangle$ |- *hole-pos* $C = s$
⟨*proof*⟩

lemma *ctxt-at-pos-ctxt-apply-hole-poss* [*simp*]: *ctxt-at-pos* $C\langle s\rangle$ (*hole-pos* C) = C
⟨*proof*⟩

abbreviation *map-funs-ctxt f* ≡ *map-ctxt f* (λ *x*. *x*)
lemma *map-funs-term-ctxt-apply* [*simp*]:
map-funs-term f $C\langle s\rangle$ = (*map-funs-ctxt f C*)⟨*map-funs-term f s*⟩
⟨*proof*⟩

lemma *map-funs-term-ctxt-decomp*:
assumes *map-funs-term fg t* = $C\langle s\rangle$
shows ∃ *D u*. *C* = *map-funs-ctxt fg D* ∧ *s* = *map-funs-term fg u* ∧ *t* = $D\langle u\rangle$
⟨*proof*⟩

lemma *prod-automata-from-none-root-dec*:
assumes *gta-lang Q* $\mathcal{A}$ ⊆ {*gpair s t*| *s t*. *funas-gterm s* ⊆ $\mathcal{F}$ ∧ *funas-gterm t* ⊆ $\mathcal{F}$}
and *q* |∈| *ta-der* $\mathcal{A}$ (*term-of-gterm t*) **and** *fst* (*groot-sym t*) = *None*
and $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$ **and** *q* |∈| *ta-productive Q* $\mathcal{A}$
shows ∃ *u*. *t* = *gterm-to-None-Some u* ∧ *funas-gterm u* ⊆ $\mathcal{F}$
⟨*proof*⟩

lemma *infinite-set-dec-infinite*:
assumes *infinite S* **and** ⋀ *s*. *s* ∈ *S* ⟹ ∃ *t*. *f t* = *s* ∧ *P t*
shows *infinite* {*t* | *t s*. *s* ∈ *S* ∧ *f t* = *s* ∧ *P t*} (**is** *infinite ?T*)
⟨*proof*⟩

lemma *Q-inf-exec-impl-Q-inf*:
assumes *gta-lang Q* $\mathcal{A}$ ⊆ {*gpair s t*| *s t*. *funas-gterm s* ⊆ *fset* $\mathcal{F}$ ∧ *funas-gterm t* ⊆ *fset* $\mathcal{F}$}
and $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-reachable* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ |⊆| *ta-productive Q* $\mathcal{A}$
and *q* ∈ *Q-inf-e* $\mathcal{A}$
shows *q* |∈| *Q-infty* $\mathcal{A}$ $\mathcal{F}$
⟨*proof*⟩

lemma *Q-inf-impl-Q-inf-exec*:
assumes *q* |∈| *Q-infty* $\mathcal{A}$ $\mathcal{F}$
shows *q* ∈ *Q-inf-e* $\mathcal{A}$
⟨*proof*⟩

lemma *Q-infty-fQ-inf-e-conv*:
assumes *gta-lang Q* $\mathcal{A}$ ⊆ {*gpair s t*| *s t*. *funas-gterm s* ⊆ *fset* $\mathcal{F}$ ∧ *funas-gterm*

$t \subseteq$ *fset* $\mathcal{F}\}$
  and $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* $\mathcal{A}$ **and** $\mathcal{Q}$ $\mathcal{A}$ $|\subseteq|$ *ta-productive* Q $\mathcal{A}$
  shows *Q-infty* $\mathcal{A}$ $\mathcal{F}$ = *fQ-inf-e* $\mathcal{A}$
  ⟨*proof*⟩

definition *Inf-reg-impl* **where**
  Inf-reg-impl R = *Inf-reg R* (*fQ-inf-e* (*ta R*))

lemma *Inf-reg-impl-sound*:
  assumes $\mathcal{L}$ $\mathcal{A}$ $\subseteq$ {*gpair s t*| *s t*. *funas-gterm s* $\subseteq$ *fset* $\mathcal{F}$ $\wedge$ *funas-gterm t* $\subseteq$ *fset* $\mathcal{F}$}
  and $\mathcal{Q}_r$ $\mathcal{A}$ $|\subseteq|$ *ta-reachable* (*ta* $\mathcal{A}$) **and** $\mathcal{Q}_r$ $\mathcal{A}$ $|\subseteq|$ *ta-productive* (*fin* $\mathcal{A}$) (*ta* $\mathcal{A}$)
  shows $\mathcal{L}$ (*Inf-reg-impl* $\mathcal{A}$) = $\mathcal{L}$ (*Inf-reg* $\mathcal{A}$ (*Q-infty* (*ta* $\mathcal{A}$) $\mathcal{F}$))
  ⟨*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 Δ Δ_E $\equiv$ *TA* (*fset-of-list* Δ) (*fset-of-list* Δ_E)

10 Computing the epsilon transitions for the composition of GTT's

definition Δ_ε*-rules* :: ('*q*, '*f*) *ta* $\Rightarrow$ ('*q*, '*f*) *ta* $\Rightarrow$ ('*q* $\times$ '*q*) *horn set* **where**
  Δ_ε*-rules A B* =
    {*zip ps qs* $\rightarrow_h$ (*p*, *q*) |*f ps p qs q*. *f ps* $\rightarrow$ *p* $|\in|$ *rules A* $\wedge$ *f qs* $\rightarrow$ *q* $|\in|$ *rules B* $\wedge$ *length ps* = *length qs*} $\cup$
    {[(*p*, *q*)] $\rightarrow_h$ (*p'*, *q*) |*p p' q*. (*p*, *p'*) $|\in|$ *eps A*} $\cup$
    {[(*p*, *q*)] $\rightarrow_h$ (*p*, *q'*) |*p q q'*. (*q*, *q'*) $|\in|$ *eps B*}

locale Δ_ε*-horn* =
  fixes *A* :: ('*q*, '*f*) *ta* **and** *B* :: ('*q*, '*f*) *ta*
begin

sublocale *horn* Δ_ε*-rules A B* ⟨*proof*⟩

lemma Δ_ε*-infer0*:
  infer0 = {(*p*, *q*) |*f p q*. *f* [] $\rightarrow$ *p* $|\in|$ *rules A* $\wedge$ *f* [] $\rightarrow$ *q* $|\in|$ *rules B*}
  ⟨*proof*⟩

lemma Δ_ε*-infer1*:
  infer1 pq X = {(*p*, *q*) |*f ps p qs q*. *f ps* $\rightarrow$ *p* $|\in|$ *rules A* $\wedge$ *f qs* $\rightarrow$ *q* $|\in|$ *rules B* $\wedge$

$length\ ps = length\ qs \wedge$
$(fst\ pq,\ snd\ pq) \in set\ (zip\ ps\ qs) \wedge set\ (zip\ ps\ qs) \subseteq insert\ pq\ X\} \cup$
$\{(p',\ snd\ pq)\ |p\ p'.\ (p,\ p')\ |{\in}|\ eps\ A \wedge p = fst\ pq\} \cup$
$\{(fst\ pq,\ q')\ |q\ q'.\ (q,\ q')\ |{\in}|\ eps\ B \wedge q = snd\ pq\}$
⟨*proof*⟩

lemma Δ_ε-*sound*:
Δ_ε-*set* $A\ B = saturate$
⟨*proof*⟩

end

11 Computing the epsilon transitions for the transitive closure of GTT's

definition Δ-*trancl-rules* :: $('q,\ 'f)\ ta \Rightarrow ('q,\ 'f)\ ta \Rightarrow ('q \times 'q)\ horn\ set$ **where**
Δ-*trancl-rules* $A\ B =$
Δ_ε-*rules* $A\ B \cup \{[(p,\ q),\ (q,\ r)] \rightarrow_h (p,\ r)\ |p\ q\ r.\ True\}$

locale Δ-*trancl-horn* =
fixes $A :: ('q,\ 'f)\ ta$ **and** $B :: ('q,\ 'f)\ ta$
begin

sublocale *horn* Δ-*trancl-rules* $A\ B$ ⟨*proof*⟩

lemma Δ-*trancl-infer0*:
$infer0 = horn.infer0\ (\Delta_\varepsilon\text{-}rules\ A\ B)$
⟨*proof*⟩

lemma Δ-*trancl-infer1*:
$infer1\ pq\ X = horn.infer1\ (\Delta_\varepsilon\text{-}rules\ A\ B)\ pq\ X \cup$
$\{(r,\ snd\ pq)\ |r\ p'.\ (r,\ p') \in X \wedge p' = fst\ pq\} \cup$
$\{(fst\ pq,\ r)\ |q'\ r.\ (q',\ r) \in (insert\ pq\ X) \wedge q' = snd\ pq\}$
⟨*proof*⟩

lemma Δ-*trancl-sound*:
Δ-*trancl-set* $A\ B = saturate$
⟨*proof*⟩

end

12 Computing the epsilon transitions for the transitive closure of pair automata

definition Δ-*Atr-rules* :: $('q \times 'q)\ fset \Rightarrow ('q,\ 'f)\ ta \Rightarrow ('q,\ 'f)\ ta \Rightarrow ('q \times 'q)$ *horn set* **where**
Δ-*Atr-rules* $Q\ A\ B =$

$\{[] \rightarrow_h (p, q) \mid p\ q.\ (p\ ,\ q)\ |{\in}|\ Q\} \cup$
$\{[(p, q),(r, v)] \rightarrow_h (p, v)\ |p\ q\ r\ v.\ (q, r)\ |{\in}|\ \Delta_\varepsilon\ B\ A\}$

locale Δ-*Atr-horn* =
fixes Q :: $('q \times 'q)$ *fset* **and** A :: $('q, 'f)$ *ta* **and** B :: $('q, 'f)$ *ta*
begin

sublocale *horn* Δ-*Atr-rules Q A B* ⟨*proof*⟩

lemma Δ-*Atr-infer0*: *infer0* = *fset Q*
⟨*proof*⟩

lemma Δ-*Atr-infer1*:
$infer1\ pq\ X = \{(p, snd\ pq) \mid p\ q.\ (p, q) \in X \wedge (q, fst\ pq)\ |{\in}|\ \Delta_\varepsilon\ B\ A\} \cup$
$\{(fst\ pq, v) \mid q\ r\ v.\ (snd\ pq, r)\ |{\in}|\ \Delta_\varepsilon\ B\ A \wedge (r, v) \in X\} \cup$
$\{(fst\ pq, snd\ pq) \mid q\ .\ (snd\ pq, fst\ pq)\ |{\in}|\ \Delta_\varepsilon\ B\ A\}$
⟨*proof*⟩

lemma Δ-*Atr-sound*:
Δ-*Atrans-set Q A B* = *saturate*
⟨*proof*⟩

end

13 Computing the Q infinity set for the infinity predicate automaton

definition *Q-inf-rules* :: $('q, 'f\ option \times 'g\ option)\ ta \Rightarrow ('q \times 'q)\ horn\ set$ **where**
Q-inf-rules A =
$\{[] \rightarrow_h (ps\ !\ i, p)\ |f\ ps\ p\ i.\ (None, Some\ f)\ ps \rightarrow p\ |{\in}|\ rules\ A \wedge i < length\ ps\} \cup$
$\{[(p, q)] \rightarrow_h (p, r)\ |p\ q\ r.\ (q, r)\ |{\in}|\ eps\ A\} \cup$
$\{[(p, q), (q, r)] \rightarrow_h (p, r)\ |p\ q\ r.\ True\}$

locale *Q-horn* =
fixes A :: $('q, 'f\ option \times 'g\ option)$ *ta*
begin

sublocale *horn Q-inf-rules A* ⟨*proof*⟩

lemma *Q-infer0*:
$infer0 = \{(ps\ !\ i, p)\ |f\ ps\ p\ i.\ (None, Some\ f)\ ps \rightarrow p\ |{\in}|\ rules\ A \wedge i < length\ ps\}$
⟨*proof*⟩

lemma *Q-infer1*:
$infer1\ pq\ X = \{(fst\ pq, r) \mid q\ r.\ (q, r)\ |{\in}|\ eps\ A \wedge q = snd\ pq\} \cup$
$\{(r, snd\ pq)\ |r\ p'.\ (r, p') \in X \wedge p' = fst\ pq\} \cup$

$\{(fst\ pq,\ r)\ |q'\ r.\ (q',\ r) \in (insert\ pq\ X) \wedge q' = snd\ pq\}$
⟨*proof*⟩

lemma *Q-sound*:
Q-inf A = saturate
⟨*proof*⟩

end

end
theory *Regular-Relation-Impl*
imports *Tree-Automata-Impl*
Regular-Relation-Abstract-Impl
Horn-Fset
begin

14 Computing the epsilon transitions for the composition of GTT's

definition Δ_ε*-infer0-cont* **where**
Δ_ε*-infer0-cont* $\Delta_A\ \Delta_B$ =
(*let arules* = *filter* (λ *r. r-lhs-states r* = []) (*sorted-list-of-fset* Δ_A) *in*
let brules = *filter* (λ *r. r-lhs-states r* = []) (*sorted-list-of-fset* Δ_B) *in*
(*map* (*map-prod r-rhs r-rhs*) (*filter* (λ(*ra*, *rb*). *r-root ra* = *r-root rb*) (*List.product arules brules*))))

definition Δ_ε*-infer1-cont* **where**
Δ_ε*-infer1-cont* $\Delta_A\ \Delta_{A\varepsilon}\ \Delta_B\ \Delta_{B\varepsilon}$ =
(*let* (*arules*, *aeps*) = (*sorted-list-of-fset* Δ_A, *sorted-list-of-fset* $\Delta_{A\varepsilon}$) *in*
let (*brules*, *beps*) = (*sorted-list-of-fset* Δ_B, *sorted-list-of-fset* $\Delta_{B\varepsilon}$) *in*
let prules = *List.product arules brules in*
(λ *pq bs*.
map (*map-prod r-rhs r-rhs*) (*filter* (λ(*ra*, *rb*). *case* (*ra*, *rb*) *of* (*TA-rule f ps p*, *TA-rule g qs q*) $\Rightarrow$
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 (λ(*p*, *p'*). (*p'*, *snd pq*)) (*filter* (λ(*p*, *p'*) $\Rightarrow$ *p* = *fst pq*) *aeps*) @
map (λ(*q*, *q'*). (*fst pq*, *q'*)) (*filter* (λ(*q*, *q'*) $\Rightarrow$ *q* = *snd pq*) *beps*)))

locale Δ_ε*-fset* =
fixes Δ_A :: (*'q* :: *linorder*, *'f* :: *linorder*) *ta-rule fset* **and** $\Delta_{A\varepsilon}$ :: (*'q* × *'q*) *fset*
and Δ_B :: (*'q*, *'f*) *ta-rule fset* **and** $\Delta_{B\varepsilon}$ :: (*'q* × *'q*) *fset*
begin

abbreviation *A* **where** $A \equiv TA\ \Delta_A\ \Delta_{A\varepsilon}$
abbreviation *B* **where** $B \equiv TA\ \Delta_B\ \Delta_{B\varepsilon}$

sublocale Δ_ε-*horn A B* ⟨*proof*⟩

sublocale *l*: *horn-fset* Δ_ε-*rules A B* Δ_ε-*infer0-cont* Δ_A Δ_B Δ_ε-*infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$
⟨*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 Δ_ε-*impl* **where**
Δ_ε-*impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *horn-fset-impl.saturate-impl* (Δ_ε-*infer0-cont* Δ_A Δ_B) (Δ_ε-*infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)

lemma Δ_ε-*impl-sound*:
assumes Δ_ε-*impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *Some xs*
shows *xs* = Δ_ε (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$)
⟨*proof*⟩

lemma Δ_ε-*impl-complete*:
fixes Δ_A :: ('*q* :: *linorder*, '*f* :: *linorder*) *ta-rule fset* **and** Δ_B :: ('*q*, '*f*) *ta-rule fset*
and $\Delta_{\varepsilon A}$ :: ('*q* × '*q*) *fset* **and** $\Delta_{\varepsilon B}$ :: ('*q* × '*q*) *fset*
shows Δ_ε-*impl* Δ_A $\Delta_{\varepsilon A}$ Δ_B $\Delta_{\varepsilon B}$ ≠ *None* ⟨*proof*⟩

lemma Δ_ε-*impl* [*code*]:
Δ_ε (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = *the* (Δ_ε-*impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)
⟨*proof*⟩

15 Computing the epsilon transitions for the transitive closure of GTT's

definition Δ-*trancl-infer0* **where**
Δ-*trancl-infer0* Δ_A Δ_B = Δ_ε-*infer0-cont* Δ_A Δ_B

definition Δ-*trancl-infer1* :: ('*q* :: *linorder* , '*f* :: *linorder*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset* ⇒ ('*q*, '*f*) *ta-rule fset* ⇒ ('*q* × '*q*) *fset*
⇒ '*q* × '*q* ⇒ ('*q* × '*q*) *fset* ⇒ ('*q* × '*q*) *list* **where**
Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ *pq bs* =
Δ_ε-*infer1-cont* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ *pq bs* @
sorted-list-of-fset (
(λ(*r*, *p*'). (*r*, *snd pq*)) |'| (*ffilter* (λ(*r*, *p*') ⇒ *p*' = *fst pq*) *bs*) |∪|
(λ(*q*', *r*). (*fst pq*, *r*)) |'| (*ffilter* (λ(*q*', *r*) ⇒ *q*' = *snd pq*) (*finsert pq bs*)))

locale Δ-*trancl-list* =

fixes Δ_A :: ('q :: linorder, 'f :: linorder) ta-rule fset **and** $\Delta_{A\varepsilon}$ :: ('q × 'q) fset
and Δ_B :: ('q, 'f) ta-rule fset **and** $\Delta_{B\varepsilon}$ :: ('q × 'q) fset
begin

abbreviation *A* **where** $A \equiv TA\ \Delta_A\ \Delta_{A\varepsilon}$
abbreviation *B* **where** $B \equiv TA\ \Delta_B\ \Delta_{B\varepsilon}$

sublocale Δ-*trancl-horn A B* ⟨*proof*⟩

sublocale *l*: *horn-fset* Δ-*trancl-rules A B*
Δ-*trancl-infer0* Δ_A Δ_B Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$
⟨*proof*⟩

lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
horn-fset-impl.saturate-impl (Δ-*trancl-infer0* Δ_A Δ_B) (Δ-*trancl-infer1* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)

lemma Δ-*trancl-impl-sound*:
assumes Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *Some xs*
shows *xs* = Δ-*trancl* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$)
⟨*proof*⟩

lemma Δ-*trancl-impl-complete*:
fixes Δ_A :: ('q :: linorder, 'f :: linorder) ta-rule fset **and** Δ_B :: ('q, 'f) ta-rule fset
and $\Delta_{A\varepsilon}$ :: ('q × 'q) fset **and** $\Delta_{B\varepsilon}$ :: ('q × 'q) fset
shows Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ ≠ *None*
⟨*proof*⟩

lemma Δ-*trancl-impl* [*code*]:
Δ-*trancl* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = (*the* (Δ-*trancl-impl* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$))
⟨*proof*⟩

16 Computing the epsilon transitions for the transitive closure of pair automata

definition Δ-*Atr-infer1-cont* :: ('q :: linorder × 'q) fset ⇒ ('q, 'f :: linorder) ta-rule fset ⇒ ('q × 'q) fset ⇒
('q, 'f) ta-rule fset ⇒ ('q × 'q) fset ⇒ 'q × 'q ⇒ ('q × 'q) fset ⇒ ('q × 'q) list
where
Δ-*Atr-infer1-cont Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
(*let G* = *sorted-list-of-fset* (*the* (Δ_ε-*impl* Δ_B $\Delta_{B\varepsilon}$ Δ_A $\Delta_{A\varepsilon}$)) *in*
(λ *pq bs*.

(*let bs-list* = *sorted-list-of-fset bs in*
map (λ (*p*, *q*). (*fst p*, *snd pq*)) (*filter* (λ (*p*, *q*). *snd p* = *fst q* $\wedge$ *snd q* = *fst pq*) (*List.product bs-list G*)) @
map (λ (*p*, *q*). (*fst pq*, *snd q*)) (*filter* (λ (*p*, *q*). *snd p* = *fst q* $\wedge$ *fst p* = *snd pq*) (*List.product G bs-list*)) @
map (λ (*p*, *q*). (*fst pq*, *snd pq*)) (*filter* (λ (*p*, *q*). *snd pq* = *p* $\wedge$ *fst pq* = *q*) *G*))))

locale Δ-*Atr-fset* =
fixes *Q* :: (*'q* :: *linorder* $\times$ *'q*) *fset* **and** Δ_A :: (*'q*, *'f* :: *linorder*) *ta-rule fset* **and** $\Delta_{A\varepsilon}$ :: (*'q* $\times$ *'q*) *fset*
and Δ_B :: (*'q*, *'f*) *ta-rule fset* **and** $\Delta_{B\varepsilon}$ :: (*'q* $\times$ *'q*) *fset*
begin

abbreviation *A* **where** *A* $\equiv$ *TA* Δ_A $\Delta_{A\varepsilon}$
abbreviation *B* **where** *B* $\equiv$ *TA* Δ_B $\Delta_{B\varepsilon}$

sublocale Δ-*Atr-horn Q A B* ⟨*proof*⟩

lemma *infer1*:
infer1 pq (*fset bs*) = *set* (Δ-*Atr-infer1-cont Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ *pq bs*)
⟨*proof*⟩

sublocale *l*: *horn-fset* Δ-*Atr-rules Q A B sorted-list-of-fset Q* Δ-*Atr-infer1-cont Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$
⟨*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 Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ =
horn-fset-impl.saturate-impl (*sorted-list-of-fset Q*) (Δ-*Atr-infer1-cont Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$)

lemma Δ-*Atr-impl-sound*:
assumes Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ = *Some xs*
shows *xs* = Δ-*Atrans Q* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$)
⟨*proof*⟩

lemma Δ-*Atr-impl-complete*:
shows Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$ $\neq$ *None* ⟨*proof*⟩

lemma Δ-*Atr-impl* [*code*]:
Δ-*Atrans Q* (*TA* Δ_A $\Delta_{A\varepsilon}$) (*TA* Δ_B $\Delta_{B\varepsilon}$) = (*the* (Δ-*Atr-impl Q* Δ_A $\Delta_{A\varepsilon}$ Δ_B $\Delta_{B\varepsilon}$))
⟨*proof*⟩

17 Computing the Q infinity set for the infinity predicate automaton

definition *Q-infer0-cont* :: ($'q$ :: *linorder*, $'f$ :: *linorder option* × $'g$ :: *linorder option*) *ta-rule fset* ⇒ ($'q$ × $'q$) *list* **where**
Q-infer0-cont Δ = *concat* (*sorted-list-of-fset* (
(λ *r*. *case r of TA-rule f ps p* ⇒ *map* (λ *x*. *Pair x p*) *ps*) |'|
(*ffilter* (λ *r*. *case r of TA-rule f ps p* ⇒ *fst f* = *None* ∧ *snd f* ≠ *None* ∧ *ps* ≠ []) Δ)))

definition *Q-infer1-cont* :: ($'q$ ::*linorder* × $'q$) *fset* ⇒ $'q$ × $'q$ ⇒ ($'q$ × $'q$) *fset* ⇒ ($'q$ × $'q$) *list* **where**
Q-infer1-cont Δε =
(*let eps* = *sorted-list-of-fset* Δε *in*
(λ *pq bs*.
let bs-list = *sorted-list-of-fset bs in*
map (λ (*q*, *r*). (*fst pq*, *r*)) (*filter* (λ (*q*, *r*) ⇒ *q* = *snd pq*) *eps*) @
map (λ(*r*, *p'*). (*r*, *snd pq*)) (*filter* (λ(*r*, *p'*) ⇒ *p'* = *fst pq*) *bs-list*) @
map (λ(*q'*, *r*). (*fst pq*, *r*)) (*filter* (λ(*q'*, *r*) ⇒ *q'* = *snd pq*) (*pq* # *bs-list*))))

locale *Q-fset* =
fixes Δ :: ($'q$ :: *linorder*, $'f$ :: *linorder option* × $'g$ :: *linorder option*) *ta-rule fset*
and Δε :: ($'q$ × $'q$) *fset*
begin

abbreviation *A* **where** *A* ≡ *TA* Δ Δε
sublocale *Q-horn A* ⟨*proof*⟩

sublocale *l*: *horn-fset Q-inf-rules A Q-infer0-cont* Δ *Q-infer1-cont* Δε
⟨*proof*⟩

lemmas *saturate-impl-sound* = *l.saturate-impl-sound*
lemmas *saturate-impl-complete* = *l.saturate-impl-complete*

end

definition *Q-impl* **where**
Q-impl Δ Δε = *horn-fset-impl.saturate-impl* (*Q-infer0-cont* Δ) (*Q-infer1-cont* Δε)

lemma *Q-impl-sound*:
Q-impl Δ Δε = *Some xs* ⟹ *fset xs* = *Q-inf* (*TA* Δ Δε)
⟨*proof*⟩

lemma *Q-impl-complete*:
Q-impl Δ Δε ≠ *None*
⟨*proof*⟩

definition *Q-infinity-impl* Δ $\Delta\varepsilon$ = (*let Q* = *the* (*Q-impl* Δ $\Delta\varepsilon$) *in*
snd |'| ((*ffilter* (λ (*p*, *q*). *p* = *q*) *Q*) |*O*| *Q*))

lemma *Q-infinity-impl-fmember*:
q |∈| *Q-infinity-impl* Δ $\Delta\varepsilon$ ⟷ (∃ *p*. (*p*, *p*) |∈| *the* (*Q-impl* Δ $\Delta\varepsilon$) ∧
(*p*, *q*) |∈| *the* (*Q-impl* Δ $\Delta\varepsilon$))
⟨*proof*⟩

lemma *loop-sound-correct* [*simp*]:
fset (*Q-infinity-impl* Δ $\Delta\varepsilon$) = *Q-inf-e* (*TA* Δ $\Delta\varepsilon$)
⟨*proof*⟩

lemma *fQ-inf-e-code* [*code*]:
fQ-inf-e (*TA* Δ $\Delta\varepsilon$) = *Q-infinity-impl* Δ $\Delta\varepsilon$
⟨*proof*⟩

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. Hriţcu and A. Popescu, editors, *Proc. 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 250–263, 2021.