Tree Automata

Peter Lammich

April 20, 2020

Abstract

This work presents a machine-checked tree automata library for Standard-ML, OCaml and Haskell. The algorithms are efficient by using appropriate data structures like RB-trees. The available algorithms for non-deterministic automata include membership query, reduction, intersection, union, and emptiness check with computation of a witness for non-emptiness.

The executable algorithms are derived from less-concrete, non-executable algorithms using data-refinement techniques. The concrete data structures are from the Isabelle Collections Framework.

Moreover, this work contains a formalization of the class of tree-regular languages and its closure properties under set operations.

Contents

1 Introduction

This work presents a tree automata library for Isabelle/HOL. Using the code-generator of Isabelle/HOL, efficient code for all supported target languages can be generated. Currently, code for Standard-ML, OCaml and Haskell is generated.

By using appropriate data structures from the Isabelle Collections Framework[4], the algorithms are rather efficient. For some (non-representative) test set (cf. Section 6.1), the Haskell-versions of the algorithms where only about 2-3 times slower than a Java-implementation, and several orders of magnitude faster than the TAML-library [3], that is implemented in OCaml.

The standard-algorithms for non-deterministic tree-automata are available, i.e. membership query, reduction[1], intersection, union, and emptiness check with computation of a witness for non-emptiness. The choice of the formalized algorithms was motivated by the requirements for a model-checker for DPNs[1], that the author is currently working on[5]. There, only intersection and emptiness check are needed, and a witness for non-emptiness is needed to derive an error-trace.

The algorithms are first formalized using the appropriate Isabelle data-types and specification mechanisms, mainly sets and inductive predicates. However, those algorithms are not efficiently executable. Hence, in a second step, those algorithms are systematically refined to use more efficient data structures from the Isabelle Collections Framework [4].

Apart from the executable algorithms, the library also contains a formalization of the class of ranked tree-regular languages and its standard closure properties. Closure under union, intersection, complement and difference is shown.

For an introduction to tree automata and the algorithms used here, see the TATA-book [2].

1.1 Submission Structure

In this section, we give a brief overview of the structure of this submission and a description of each file and directory.

1.1.1 common/

This directory contains a collection of generally useful theories.

Misc.thy Collection of various lemmas augmenting isabelle's standard library.

[1] Currently only backward (utility) reduction is refined to executable code

1.1.2 common/bugfixes/

This directory contains bugfixes of the Isabelle standard libraries and tools. Currently, just one fix for the OCaml code-generator.

Efficient_Nat.thy Replaces *Library/Efficient_Nat.thy*. Fixes issue with OCaml code generation. Provided by Florian Haftmann.

1.1.3 ./

This is the main directory of the submission, and contains the formalization of tree automata.

AbsAlgo.thy Algorithms on tree automata.

Ta_impl.thy Executable implementation of tree automata.

Ta.thy Formalization of tree automata and basic properties.

Tree.thy Formalization of trees.

document/ Contains files for latex document creation

IsaMakefile Isabelle makefile to check the proofs and build logic image and latex documents

ROOT.ML Setup for theories to be proofchecked and included into latex documents

TODO Todo list

1.1.4 code/

This directory contains the generated code as well as some test cases for performance measurement.

The test-cases consists of pairs of medium-sized tree automata (10-100 states, a few hundred rules). The performance test intersects the automata from each pair and checks the result for emptiness. If the result is not-empty, a tree accepted by both automata is constructed.

Currently, the tests are restricted to finding witnesses of non-emptiness for intersection, as this is the intended application of this library by the author.

doTests.sh Shell-script to compile all test-cases and start the performance measurement. When finnished, the script outputs an overview of the time needed by all supported languages.

1.1.5 code/ml/

This directory contains the SML code.

code/ml/generated/ Contains the file *Ta.ML*, created by Isabelle's code generator. This file declares a module *Ta* that contains all functions of the tree automata interface.

doTests.sh Shell script to execute SML performance test

Main.ML This file executes the ML performance tests.

pt_examples.ML This file contains the input data for the performance test.

run.sh Used by doTests.sh

test_setup.ML Required by *Main.ML*

1.1.6 code/ocaml/

This directory contains the OCaml code.

code/ocaml/generated/ Contains the file *Ta.ml*, created by Isabelle's code generator. This file declares a module *Ta* that contains all functions of the tree automata interface.

doTests.sh Shell script to compile and execute OCaml performance test.

Main.ml Main file for compiled performance tests.

Main_script.ml Main file for scripted performance tests.

make.sh Compile performance test files.

Pt_examples.ml Contains the input data for the performance test.

run_script.sh Run the performance test in script mode (slow).

Test_setup.ml Required by *Main.ml* and *Main_script.ml*.

1.1.7 code/haskell/

This directory contains the Haskell code.

code/haskell/generated/ Contains the files generated by Isabelle's code generator. The *Ta.hs* declares the module *Ta* that contains the tree automata interface. There may be more files in this directory, that declare modules that are imported by *Ta*.

doTests.sh Compile and execute performance tests.

Main.hs Source-code of performance tests.

make.sh Compile performance tests.

Pt_examples.hs Input data for performance tests.

1.1.8 code/taml/

This directory contains the Timbuk/Taml test cases.

Main.ml Runs the test-cases. To be executed within the Taml-toplevel.

code/taml/tests/ This directory contains Taml input files for the test cases.

2 Trees

theory *Tree*
imports *Main*
begin

This theory defines trees as nodes with a label and a list of subtrees.

datatype *'l tree = NODE 'l 'l tree list*

datatype-compat *tree*

end

3 Tree Automata

theory *Ta*
imports *Main Automatic-Refinement.Misc Tree*
begin

This theory defines tree automata, tree regular languages and specifies basic algorithms.

Nondeterministic and deterministic (bottom-up) tree automata are defined.

For non-deterministic tree automata, basic algorithms for membership, union, intersection, forward and backward reduction, and emptiness check are specified. Moreover, a (brute-force) determinization algorithm is specified.

For deterministic tree automata, we specify algorithms for complement and completion.

Finally, the class of regular languages over a given ranked alphabet is defined and its standard closure properties are proved.

The specification of the algorithms in this theory is very high-level, and the specifications are not executable. A bit more specific algorithms are defined in Section 4, and a refinement to executable definitions is done in Section 5.

3.1 Basic Definitions

3.1.1 Tree Automata

A tree automata consists of a (finite) set of initial states and a (finite) set of rules.

A rule has the form $q \to l\ q1 \ldots qn$, with the meaning that one can derive $l(q1 \ldots qn)$ from the state q.

datatype $('q,'l)$ *ta-rule* = *RULE* $'q$ $'l$ $'q$ *list* (- → - -)

record $('Q,'L)$ *tree-automaton-rec* =
  ta-initial :: $'Q$ *set*
  ta-rules :: $('Q,'L)$ *ta-rule set*

  — Rule deconstruction
fun *lhs* **where** *lhs* $(q \to l\ qs) = q$
fun *rhsq* **where** *rhsq* $(q \to l\ qs) = qs$
fun *rhsl* **where** *rhsl* $(q \to l\ qs) = l$
  — States in a rule
fun *rule-states* **where** *rule-states* $(q \to l\ qs)$ = *insert* q (*set* qs)
  — States in a set of rules
definition δ-*states* $\delta == \bigcup$(*rule-states* ‘ δ)
  — States in a tree automaton
definition *ta-rstates* *TA* = *ta-initial* *TA* ∪ δ-*states* (*ta-rules* *TA*)
  — Symbols occurring in rules
definition δ-*symbols* δ == *rhsl*‘δ

  — Nondeterministic, finite tree automaton (NFTA)
locale *tree-automaton* =
  fixes *TA* :: $('Q,'L)$ *tree-automaton-rec*
  assumes *finite-rules*[*simp*, *intro*!]: *finite* (*ta-rules* *TA*)
  assumes *finite-initial*[*simp*, *intro*!]: *finite* (*ta-initial* *TA*)
begin
  abbreviation *Qi* == *ta-initial* *TA*
  abbreviation δ == *ta-rules* *TA*
  abbreviation *Q* == *ta-rstates* *TA*
end

3.1.2 Acceptance

The predicate *accs* δ t q is true, iff the tree t is accepted in state q w.r.t. the rules in δ.

A tree is accepted in state q, if it can be produced from q using the rules.

inductive *accs* :: ('*Q*,'*L*) *ta-rule set* ⇒ '*L tree* ⇒ '*Q* ⇒ *bool*
where
⟦
(*q* → *f qs*) ∈ δ; *length ts* = *length qs*;
!!*i*. *i*<*length qs* ⟹ *accs* δ (*ts* ! *i*) (*qs* ! *i*)
⟧ ⟹ *accs* δ (*NODE f ts*) *q*

— Characterization of *accs* using *list-all-zip*
inductive *accs-laz* :: ('*Q*,'*L*) *ta-rule set* ⇒ '*L tree* ⇒ '*Q* ⇒ *bool*
where
⟦
(*q* → *f qs*) ∈ δ;
list-all-zip (*accs-laz* δ) *ts qs*
⟧ ⟹ *accs-laz* δ (*NODE f ts*) *q*

lemma *accs-laz*: *accs* = *accs-laz*
apply (*intro ext*)
apply (*rule iffI*)
apply (*erule accs.induct*)
apply (*auto intro*: *accs-laz.intros*[*simplified list-all-zip-alt*])
apply (*erule accs-laz.induct*)
apply (*auto intro*: *accs.intros simp add*: *list-all-zip-alt*)
done

3.1.3 Language

The language of a tree automaton is the set of all trees that are accepted in an initial state.

definition *ta-lang TA* == { *t* . ∃ *q*∈*ta-initial TA*. *accs* (*ta-rules TA*) *t q* }

3.2 Basic Properties

lemma *rule-states-simp*:
rule-states x = (*case x of* (*q* → *l qs*) ⇒ *insert q* (*set qs*))
by (*case-tac x*) *auto*

lemma *rule-states-lhs*[*simp*]: *lhs r* ∈ *rule-states r*
by (*auto split*: *ta-rule.split simp add*: *rule-states-simp*)

lemma *rule-states-rhsq*: *set* (*rhsq r*) ⊆ *rule-states r*
by (*auto split*: *ta-rule.split simp add*: *rule-states-simp*)

lemma *rule-states-finite*[*simp*, *intro*!]: *finite* (*rule-states r*)
by (*simp add*: *rule-states-simp split*: *ta-rule.split*)

lemma δ-*statesI*:
assumes *A*: (*q* → *l qs*)∈δ
shows *q*∈δ-*states* δ

set qs $\subseteq$ *δ-states δ*
using *A*
apply (*unfold δ-states-def*)
by (*auto split*: *ta-rule.split simp add*: *rule-states-simp*)

lemma *δ-statesI'*: ⟦(*q* → *l qs*)∈*δ*; *qi*∈*set qs*⟧ ⟹ *qi*∈*δ-states δ*
using *δ-statesI*(*2*) **by** *fast*

lemma *δ-states-accsI*: *accs δ n q* ⟹ *q*∈*δ-states δ*
by (*auto elim*: *accs.cases intro*: *δ-statesI*)

lemma *δ-states-union*[*simp*]: *δ-states* (*δ*∪*δ'*) = *δ-states δ* ∪ *δ-states δ'*
by (*auto simp add*: *δ-states-def*)

lemma *δ-states-insert*[*simp*]:
δ-states (*insert r δ*) = (*rule-states r* ∪ *δ-states δ*)
by (*unfold δ-states-def*) *auto*

lemma *δ-states-mono*: ⟦*δ* ⊆ *δ'*⟧ ⟹ *δ-states δ* ⊆ *δ-states δ'*
by (*unfold δ-states-def*) *auto*

lemma *δ-states-finite*[*simp*, *intro*]: *finite δ* ⟹ *finite* (*δ-states δ*)
by (*unfold δ-states-def*) *auto*

lemma *δ-statesE*: ⟦*q*∈*δ-states* Δ;
!!*f qs*. ⟦ (*q* → *f qs*)∈Δ ⟧ ⟹ *P*;
!!*ql f qs*. ⟦ (*ql* → *f qs*)∈Δ; *q*∈*set qs* ⟧ ⟹ *P*
⟧ ⟹ *P*
apply (*unfold δ-states-def*)
apply (*auto*)
apply (*auto simp add*: *rule-states-simp split*: *ta-rule.split-asm*)
done

lemma *δ-symbolsI*: (*q* → *f qs*)∈*δ* ⟹ *f*∈*δ-symbols δ*
by (*force simp add*: *δ-symbols-def*)

lemma *δ-symbolsE*:
assumes *A*: *f*∈*δ-symbols δ*
obtains *q qs* **where** (*q* → *f qs*) ∈ *δ*
using *A*
apply (*simp add*: *δ-symbols-def*)
apply (*erule imageE*)
apply (*case-tac x*)
apply *simp*
done

lemma *δ-symbols-simps*[*simp*]:
δ-symbols {} = {}
δ-symbols (*insert r δ*) = *insert* (*rhsl r*) (*δ-symbols δ*)

δ-symbols $(\delta \cup \delta') = \delta$*-symbols* $\delta \cup \delta$*-symbols* δ'
by (*auto simp add*: *δ-symbols-def*)

lemma *δ-symbols-finite*[*simp*, *intro*!]:
finite $\delta \Longrightarrow$ *finite* (*δ-symbols* δ)
by (*auto simp add*: *δ-symbols-def*)

lemma *accs-mono*: ⟦*accs* δ *n q*; $\delta \subseteq \delta'$⟧ $\Longrightarrow$ *accs* δ' *n q*
proof (*induct rule*: *accs.induct*[*case-names step*])
case (*step q l qs* δ *n*)
hence *R'*: $(q \to l\ qs) \in \delta'$ **by** *auto*
from *accs.intros*[*OF R' step.hyps*(*2*)]
step.hyps(*4*)[*OF - step.prems*]
show *?case* **.**
qed

context *tree-automaton*
begin
lemma *initial-subset*: *ta-initial TA* $\subseteq$ *ta-rstates TA*
by (*unfold ta-rstates-def*) *auto*
lemma *states-subset*: *δ-states* (*ta-rules TA*) $\subseteq$ *ta-rstates TA*
by (*unfold ta-rstates-def*) *auto*

lemma *finite-states*[*simp*, *intro*!]: *finite* (*ta-rstates TA*)
by (*auto simp add*: *ta-rstates-def δ-states-def*
intro: *finite-rules finite-UN-I*)

lemma *finite-symbols*[*simp*, *intro*!]: *finite* (*δ-symbols* (*ta-rules TA*))
by *simp*

lemmas *is-subset* = *rev-subsetD*[*OF - initial-subset*]
rev-subsetD[*OF - states-subset*]
end

3.3 Other Classes of Tree Automata

3.3.1 Automata over Ranked Alphabets

— All trees over ranked alphabet
inductive-set *ranked-trees* :: (*'L* ⇀ *nat*) ⇒ *'L tree set*
for *A* **where**
⟦ ∀ *t*∈*set ts*. *t*∈*ranked-trees A*; *A f* = *Some* (*length ts*) ⟧
$\Longrightarrow$ *NODE f ts* ∈ *ranked-trees A*

locale *finite-alphabet* =
fixes *A* :: (*'L* ⇀ *nat*)
assumes *A-finite*[*simp*, *intro*!]: *finite* (*dom A*)
begin
abbreviation *F* == *dom A*
end

context *finite-alphabet*
begin

definition *legal-rules Q* == { (*q* → *f qs*) | *q f qs*.
q ∈ *Q*
∧ *qs* ∈ *lists Q*
∧ *A f* = *Some* (*length qs*)}

lemma *legal-rulesI*:
⟦
r∈δ;
rule-states r ⊆ *Q*;
A (*rhsl r*) = *Some* (*length* (*rhsq r*))
⟧ ⟹ *r*∈*legal-rules Q*
apply (*unfold legal-rules-def*)
apply (*cases r*)
apply (*auto*)
done

lemma *legal-rules-finite*[*simp*, *intro*!]:
fixes *Q*::'*Q set*
assumes [*simp*, *intro*!]: *finite Q*
shows *finite* (*legal-rules Q*)
proof −
define *possible-rules-f*
where *possible-rules-f* = (λ(*Q*::'*Q set*) *f*.
(λ(*q*,*qs*). (*q* → *f qs*)) ‘ (*Q* × (*lists Q* ∩ {*qs*. *A f* = *Some* (*length qs*)})))

have *legal-rules Q* = ⋃(*possible-rules-f Q*‘*F*)
by (*auto simp add*: *legal-rules-def possible-rules-f-def*)
moreover have !!*f*. *finite* (*possible-rules-f Q f*)
apply (*unfold possible-rules-f-def*)
apply (*rule finite-imageI*)
apply (*rule finite-SigmaI*)
apply *simp*
apply (*case-tac A f*)
apply *simp*
apply (*simp add*: *lists-of-len-fin*)
done
ultimately show *?thesis* **by** *auto*
qed
end

— Finite tree automata with ranked alphabet
locale *ranked-tree-automaton* =
tree-automaton TA +
finite-alphabet A
for *TA* :: ('*Q*,'*L*) *tree-automaton-rec*

and $A :: 'L \rightharpoonup nat$ +
assumes *ranked*: $(q \to f\ qs) \in \delta \Longrightarrow A\ f = Some\ (length\ qs)$
begin

lemma *rules-legal*: $r \in \delta \Longrightarrow r \in$ *legal-rules* Q
apply (*rule legal-rulesI*)
apply *assumption*
apply (*auto simp add*: *ta-rstates-def δ-states-def*) [*1*]
apply (*case-tac r*)
apply (*auto intro*: *ranked*)
done

— Only well-ranked trees are accepted
lemma *accs-is-ranked*: *accs δ t q* $\Longrightarrow t \in$ *ranked-trees A*
apply (*induct δ≡δ t q rule*: *accs.induct*)
apply (*rule ranked-trees.intros*)
apply (*auto simp add*: *set-conv-nth ranked*)
done

— The language consists of well-ranked trees
theorem *lang-is-ranked*: *ta-lang TA* $\subseteq$ *ranked-trees A*
using *accs-is-ranked* **by** (*auto simp add*: *ta-lang-def*)

end

3.3.2 Deterministic Tree Automata

— Deterministic, (bottom-up) finite tree automaton (DFTA)
locale *det-tree-automaton* = *ranked-tree-automaton TA A*
for *TA* :: $('Q,'L)$ *tree-automaton-rec* **and** *A* +
assumes *deterministic*: $[\![(q \to f\ qs) \in \delta; (q' \to f\ qs) \in \delta\ ]\!] \Longrightarrow q = q'$
begin
theorem *accs-unique*: $[\![$ *accs δ t q*; *accs δ t q'* $]\!] \Longrightarrow q = q'$
unfolding *accs-laz*
proof (*induct δ≡δ t q arbitrary*: q' *rule*: *accs-laz.induct*[*case-names step*])
case (*step q f qs ts q'*)
hence *I*:
$(q \to f\ qs) \in \delta$
list-all-zip (*accs-laz δ*) *ts qs*
list-all-zip ($\lambda t\ q.\ (\forall q'.\ accs\text{-}laz\ \delta\ t\ q' \longrightarrow q = q')$) *ts qs*
accs-laz δ (*NODE f ts*) q'
by *auto*
from *I*(*4*) **obtain** qs' **where** A':
$(q' \to f\ qs') \in \delta$
list-all-zip (*accs-laz δ*) *ts* qs'
by (*auto elim*!: *accs-laz.cases*)

from *I*(*2*,*3*) $A'(2)$ **have** *list-all-zip* (=) *qs* qs'
by (*auto simp add*: *list-all-zip-alt*)

```
    hence qs=qs' by (auto simp add: laz-eq)
    with deterministic[OF I(1), of q'] A'(1) show q=q' by simp
  qed

end
```

3.3.3 Complete Tree Automata

```
locale complete-tree-automaton = det-tree-automaton TA A
  for TA :: ('Q,'L) tree-automaton-rec and A
  +
  assumes complete:
  ⟦ qs∈lists Q; A f = Some (length qs) ⟧ ⟹ ∃ q. (q → f qs)∈δ
begin

    — In a complete DFTA, all trees can be labeled by some state
  theorem label-all: t∈ranked-trees A ⟹ ∃ q∈Q. accs δ t q
  proof (induct rule: ranked-trees.induct[case-names constr])
    case (constr ts f)
    obtain qs where QS:
      qs∈lists Q
      list-all-zip (accs δ) ts qs
      and [simp]: length qs = length ts
    proof −
      from constr(1) have ∀ i<length ts. ∃ q. q∈Q ∧ accs δ (ts!i) q
        by (auto)
      thus ?thesis
        apply (erule-tac obtain-list-from-elements)
        apply (rule-tac that)
        apply (auto simp add: list-all-zip-alt set-conv-nth)
        done
    qed
    moreover from complete[OF QS(1), simplified, OF constr(2)] obtain q
      where (q → f qs) ∈δ ..
    ultimately show ?case
      by (auto simp add: accs-laz ta-rstates-def
            intro: accs-laz.intros δ-statesI)
  qed

end
```

3.4 Algorithms

In this section, basic algorithms on tree-automata are specified. The specification is a high-level, non-executable specification, intended to be refined to more low-level specifications, as done in Sections 4 and 5.

3.4.1 Empty Automaton

definition *ta-empty* == (| *ta-initial* = {}, *ta-rules* = {} |)

theorem *ta-empty-lang*[*simp*]: *ta-lang ta-empty* = {}
  by (*auto simp add*: *ta-empty-def ta-lang-def*)

theorem *ta-empty-ta*[*simp*, *intro*!]: *tree-automaton ta-empty*
  apply (*unfold-locales*)
  apply (*unfold ta-empty-def*)
  apply *auto*
  done

theorem (**in** *finite-alphabet*) *ta-empty-rta*[*simp*, *intro*!]:
  ranked-tree-automaton ta-empty A
  apply (*unfold-locales*)
  apply (*unfold ta-empty-def*)
  apply *auto*
  done

theorem (**in** *finite-alphabet*) *ta-empty-dta*[*simp*, *intro*!]:
  det-tree-automaton ta-empty A
  apply (*unfold-locales*)
  apply (*unfold ta-empty-def*)
  apply (*auto*)
  done

3.4.2 Remapping of States

fun *remap-rule* **where** *remap-rule f* (*q* → *l qs*) = ((*f q*) → *l* (*map f qs*))
definition
  ta-remap f TA == (| *ta-initial* = *f* ‘ *ta-initial TA*,
                  ta-rules = *remap-rule f* ‘ *ta-rules TA*
                |)

lemma *δ-states-remap*[*simp*]: *δ-states* (*remap-rule f* ‘ *δ*) = *f*‘ *δ-states δ*
  apply (*auto simp add*: *δ-states-def*)
  apply (*case-tac a*)
  apply *force*
  apply (*case-tac xb*)
  apply *force*
  done

lemma *remap-accs1*: *accs δ n q* ⟹ *accs* (*remap-rule f* ‘ *δ*) *n* (*f q*)
proof (*induct rule*: *accs.induct*[*case-names step*])
  case (*step q l qs δ ts*)
  from *step.hyps*(*1*) **have** *1*: ((*f q*) → *l* (*map f qs*)) ∈ *remap-rule f* ‘ *δ*
    by (*drule-tac f*=*remap-rule f* **in** *imageI*) *simp*
  show *?case* **proof** (*rule accs.intros*[*OF 1*])
    fix *i* **assume** *i*<*length* (*map f qs*)

```
    with step.hyps(4) show accs (remap-rule f ‘ δ) (ts ! i) (map f qs ! i)
      by auto
  qed (auto simp add: step.hyps(2))
qed

lemma remap-lang1: t∈ta-lang TA ⟹ t∈ta-lang (ta-remap f TA)
  by (unfold ta-lang-def ta-remap-def) (auto dest: remap-accs1)

lemma remap-accs2: ⟦
    accs δ' n q';
    δ'=(remap-rule f ‘ δ);
    q'=f q;
    inj-on f Q;
    q∈Q;
    δ-states δ ⊆ Q
  ⟧ ⟹ accs δ n q
proof (induct arbitrary: δ q rule: accs.induct[case-names step])
  case (step q' l qs δ' ts δ q)
  note [simp] = step.prems(1,2)
  from step.hyps(1)[simplified] step.prems(3,4,5) have
    R: (q → l (map (inv-on f Q) qs))∈δ
    apply (erule-tac imageE)
    apply (case-tac x)
    apply (auto simp del:map-map)
    apply (subst inj-on-map-inv-f)
    apply (auto dest: δ-statesI) [2]
    apply (subgoal-tac q∈δ-states δ)
    apply (unfold inj-on-def) [1]
    apply (metis δ-statesI(1) contra-subsetD)
    apply (fastforce intro: δ-statesI(1) dest: inj-onD)
    done
  show ?case proof (rule accs.intros[OF R])
    fix i
    assume i < length (map (inv-on f Q) qs)
    hence L: i<length qs by simp

    from step.hyps(1)[simplified] step.prems(5) have
      IR: !!i. i<length qs ⟹ qs!i ∈ f ‘ Q
      apply auto
      apply (case-tac x)
      apply (auto)
      apply (rename-tac list)
      apply (subgoal-tac list!i ∈ δ-states δ)
      apply blast
      apply (auto dest!: δ-statesI(2))
      done

    show accs δ (ts ! i) (map (inv-on f Q) qs ! i)
      apply (rule step.hyps(4)[OF L, simplified])
```

```
    apply (simp-all add: f-inv-on-f[OF IR[OF L]]
                         inv-on-f-range[OF IR[OF L]]
                         L step.prems(3,5))
    done
  qed (auto simp add: step.hyps(2))
qed

lemma (in tree-automaton) remap-lang2:
  assumes I: inj-on f (ta-rstates TA)
  shows t∈ta-lang (ta-remap f TA) ⟹ t∈ta-lang TA
  apply (unfold ta-lang-def ta-remap-def)
  apply auto
  apply (rule-tac x=x in bexI)
  apply (drule remap-accs2[OF - - - I])
  apply (auto dest: is-subset)
  done

theorem (in tree-automaton) remap-lang:
  inj-on f (ta-rstates TA) ⟹ ta-lang (ta-remap f TA) = ta-lang TA
  by (auto intro: remap-lang1 remap-lang2)

lemma (in tree-automaton) remap-ta[intro!, simp]:
  tree-automaton (ta-remap f TA)
  using initial-subset states-subset finite-states finite-rules
  by (unfold-locales) (auto simp add: ta-remap-def ta-rstates-def)

lemma (in ranked-tree-automaton) remap-rta[intro!, simp]:
  ranked-tree-automaton (ta-remap f TA) A
proof −
  interpret ta: tree-automaton (ta-remap f TA) by simp
  show ?thesis
    apply (unfold-locales)
    apply (auto simp add: ta-remap-def)
    apply (case-tac x)
    apply (auto simp add: ta-remap-def intro: ranked)
    done
qed

lemma (in det-tree-automaton) remap-dta[intro, simp]:
  assumes INJ: inj-on f Q
  shows det-tree-automaton (ta-remap f TA) A
proof −
  interpret ta: ranked-tree-automaton (ta-remap f TA) A by simp
  show ?thesis
  proof
    fix q q' l qs
    assume A:
      (q → l qs) ∈ta-rules (ta-remap f TA)
      (q' → l qs) ∈ta-rules (ta-remap f TA)
```

```
    then obtain qo qo' qso qso' where RO:
      (qo → l qso) ∈ δ
      (qo' → l qso') ∈ δ
      and [simp]:
      q=f qo
      q'=f qo'
      qs = map f qso
      map f qso = map f qso'
      apply (auto simp add: ta-remap-def)
      apply (case-tac x, case-tac xa)
      apply auto
      done
    from RO have OQ: qo∈Q    qo'∈Q    set qso ⊆ Q    set qso' ⊆ Q
      by (unfold ta-rstates-def)
        (auto dest: δ-statesI)

    from OQ(3,4) have INJQSO: inj-on f (set qso ∪ set qso')
      by (auto intro: subset-inj-on[OF INJ])

    from inj-on-map-eq-map[OF INJQSO] have qso=qso' by simp
    with deterministic[OF RO(1)] RO(2) have qo=qo' by simp
    thus q=q' by simp
  qed
qed


lemma (in complete-tree-automaton) remap-cta[intro, simp]:
  assumes INJ: inj-on f Q
  shows complete-tree-automaton (ta-remap f TA) A
proof −
  interpret ta: det-tree-automaton (ta-remap f TA) A by (simp add: INJ)
  show ?thesis
  proof
    fix qs l
    assume A:
      qs ∈ lists (ta-rstates (ta-remap f TA))
      A l = Some (length qs)
    from A(1) have qs∈lists (f‘Q)
      by (auto simp add: ta-rstates-def ta-remap-def)
    then obtain qso where QSO:
      qso∈lists Q
      qs = map f qso
      by (blast elim!: lists-image-witness)
    hence [simp]: length qso = length qs by simp

    from complete[OF QSO(1)] A(2) obtain qo where (qo → l qso) ∈ δ
      by auto

    with QSO(2) have ((f qo) → l qs)∈ta-rules (ta-remap f TA)
```

by (*force simp add*: *ta-remap-def*)
thus $\exists\, q.\ q \to l\ qs \in$ *ta-rules* (*ta-remap f TA*) **..**
qed
qed

3.4.3 Union

definition *ta-union TA TA′* ==
⦇ *ta-initial* = *ta-initial TA* ∪ *ta-initial TA′*,
ta-rules = *ta-rules TA* ∪ *ta-rules TA′*
⦈

— Given two disjoint sets of states, where no rule contains states from both sets, then any accepted tree is also accepted when only using one of the subsets of states and rules. This lemma and its corollaries capture the basic idea of the union-algorithm.
lemma *accs-exclusive-aux*:
⟦ *accs δn n q*; *δn*=*δ*∪*δ′*; *δ-states δ* ∩ *δ-states δ′* = {}; *q*∈*δ-states δ* ⟧
⟹ *accs δ n q*
proof (*induct arbitrary*: *δ δ′ rule*: *accs.induct*[*case-names step*])
case (*step q l qs δn ts δ δ′*)
note [*simp*] = *step.prems*(*1*)
note [*simp*] = *step.hyps*(*2*)[*symmetric*] *step.hyps*(*3*)
from *step.prems* **have** *q*∉*δ-states δ′* **by** *blast*
with *step.hyps*(*1*) **have** *set qs* ⊆ *δ-states δ* **and** *R*: (*q* → *l qs*)∈*δ*
by (*auto dest*: *δ-statesI*)
hence !!*i*. *i*<*length qs* ⟹ *accs δ* (*ts* ! *i*) (*qs* ! *i*)
by (*force intro*: *step.hyps*(*4*)[*OF* - *step.prems*(*1*,*2*)])
with *accs.intros*[*OF R step.hyps*(*2*)] **show** *?case* **.**
qed

corollary *accs-exclusive1*:
⟦ *accs* (*δ*∪*δ′*) *n q*; *δ-states δ* ∩ *δ-states δ′* = {}; *q*∈*δ-states δ* ⟧
⟹ *accs δ n q*
using *accs-exclusive-aux*[*of* - *n q δ δ′*] **by** *blast*

corollary *accs-exclusive2*:
⟦ *accs* (*δ*∪*δ′*) *n q*; *δ-states δ* ∩ *δ-states δ′* = {}; *q*∈*δ-states δ′* ⟧
⟹ *accs δ′ n q*
using *accs-exclusive-aux*[*of* - *n q δ′ δ*] **by** *blast*

lemma *ta-union-correct-aux1*:
fixes *TA TA′*
assumes *TA*: *tree-automaton TA*
assumes *TA′*: *tree-automaton TA′*
assumes *DJ*: *ta-rstates TA* ∩ *ta-rstates TA′* = {}
shows *ta-lang* (*ta-union TA TA′*) = *ta-lang TA* ∪ *ta-lang TA′*
proof (*safe*)
interpret *ta*: *tree-automaton TA* **using** *TA* **.**

```
  interpret ta': tree-automaton TA' using TA' .

  from DJ ta.states-subset ta'.states-subset have
    DJ': δ-states (ta-rules TA) ∩ δ-states (ta-rules TA') = {}
    by blast

  fix n
  assume A: n ∈ ta-lang (ta-union TA TA')    n ∉ ta-lang TA'
  from A(1) obtain q where
    B: q∈ta-initial TA ∪ ta-initial TA'
       accs (ta-rules TA ∪ ta-rules TA') n q
    by (auto simp add: ta-lang-def ta-union-def)
  from δ-states-accsI[OF B(2), simplified] show n∈ta-lang TA proof
    assume C: q∈δ-states (ta-rules TA)
    with accs-exclusive1[OF B(2) DJ'] have accs (ta-rules TA) n q .
    moreover from DJ C ta'.initial-subset ta.states-subset B(1) have
      q∈ta-initial TA
      by auto
    ultimately show ?thesis by (unfold ta-lang-def) auto
  next
    assume C: q∈δ-states (ta-rules TA')
    with accs-exclusive2[OF B(2) DJ'] have accs (ta-rules TA') n q .
    moreover from DJ C ta.initial-subset B(1) ta'.states-subset have
      q∈ta-initial TA'
      by auto
    ultimately have False using A(2) by (unfold ta-lang-def) auto
    thus ?thesis ..
  qed
qed (unfold ta-lang-def ta-union-def, auto intro: accs-mono)

lemma ta-union-correct-aux2:
  fixes TA TA'
  assumes TA: tree-automaton TA
  assumes TA': tree-automaton TA'
  shows tree-automaton (ta-union TA TA')
proof −
  interpret ta: tree-automaton TA using TA .
  interpret ta': tree-automaton TA' using TA' .

  show ?thesis
    apply (unfold-locales)
    apply (unfold ta-union-def)
    apply auto
    done
qed

  — If the sets of states are disjoint, the language of the union-automaton is the
    union of the languages of the original automata.
theorem ta-union-correct:
```

fixes *TA TA′*
assumes *TA*: *tree-automaton TA*
assumes *TA′*: *tree-automaton TA′*
assumes *DJ*: *ta-rstates TA ∩ ta-rstates TA′ = {}*
shows *ta-lang* (*ta-union TA TA′*) = *ta-lang TA ∪ ta-lang TA′*
tree-automaton (*ta-union TA TA′*)
using *ta-union-correct-aux1*[*OF TA TA′ DJ*]
ta-union-correct-aux2[*OF TA TA′*]
by *auto*

lemma *ta-union-rta*:
fixes *TA TA′*
assumes *TA*: *ranked-tree-automaton TA A*
assumes *TA′*: *ranked-tree-automaton TA′ A*
shows *ranked-tree-automaton* (*ta-union TA TA′*) *A*
proof −
interpret *ta*: *ranked-tree-automaton TA A* **using** *TA* **.**
interpret *ta′*: *ranked-tree-automaton TA′ A* **using** *TA′* **.**

show *?thesis*
apply (*unfold-locales*)
apply (*unfold ta-union-def*)
apply (*auto intro*: *ta.ranked ta′.ranked*)
done
qed

The union-algorithm may wrap the states of the first and second automaton in order to make them disjoint

datatype (*'q1*,*'q2*) *ustate-wrapper* = *USW1 'q1* | *USW2 'q2*

lemma *usw-disjoint*[*simp*]:
USW1 ‘ X ∩ USW2 ‘ Y = {}
remap-rule USW1 ‘ X ∩ remap-rule USW2 ‘ Y = {}
apply *auto*
apply (*case-tac xa*, *case-tac xb*)
apply *auto*
done

lemma *states-usw-disjoint*[*simp*]:
ta-rstates (*ta-remap USW1 X*) ∩ *ta-rstates* (*ta-remap USW2 Y*) = {}
by (*auto simp add*: *ta-remap-def ta-rstates-def*)

lemma *usw-inj-on*[*simp*, *intro*!]:
inj-on USW1 X
inj-on USW2 X
by (*auto intro*: *inj-onI*)

definition *ta-union-wrap TA TA′* =
ta-union (*ta-remap USW1 TA*) (*ta-remap USW2 TA′*)

lemma *ta-union-wrap-correct*:
fixes *TA* :: (*'Q1*,*'L*) *tree-automaton-rec*
fixes *TA'* :: (*'Q2*,*'L*) *tree-automaton-rec*
assumes *TA*: *tree-automaton TA*
assumes *TA'*: *tree-automaton TA'*
shows *ta-lang* (*ta-union-wrap TA TA'*) = *ta-lang TA* ∪ *ta-lang TA'* (**is** *?T1*)
tree-automaton (*ta-union-wrap TA TA'*) (**is** *?T2*)
proof −
interpret *a1*: *tree-automaton TA* **by** *fact*
interpret *a2*: *tree-automaton TA'* **by** *fact*

show *?T1 ?T2*
by (*unfold ta-union-wrap-def*)
(*simp-all add*: *ta-union-correct a1.remap-lang a2.remap-lang*)
qed

lemma *ta-union-wrap-rta*:
fixes *TA TA'*
assumes *TA*: *ranked-tree-automaton TA A*
assumes *TA'*: *ranked-tree-automaton TA' A*
shows *ranked-tree-automaton* (*ta-union-wrap TA TA'*) *A*
proof −
interpret *ta*: *ranked-tree-automaton TA A* **using** *TA* **.**
interpret *ta'*: *ranked-tree-automaton TA' A* **using** *TA'* **.**

show *?thesis*
by (*unfold ta-union-wrap-def*)
(*simp add*: *ta-union-rta*)

qed

3.4.4 Reduction

definition *reduce-rules δ P* == *δ* ∩ { *r*. *rule-states r* ⊆ *P* }

lemma *reduce-rulesI*: ⟦*r*∈*δ*; *rule-states r* ⊆ *P*⟧ ⟹ *r*∈*reduce-rules δ P*
by (*unfold reduce-rules-def*) *auto*

lemma *reduce-rulesD*:
⟦ *r*∈*reduce-rules δ P* ⟧ ⟹ *r*∈*δ*
⟦ *r*∈*reduce-rules δ P*; *q*∈*rule-states r*⟧ ⟹ *q*∈*P*
by (*unfold reduce-rules-def*) *auto*

lemma *reduce-rules-subset*: *reduce-rules δ P* ⊆ *δ*
by (*unfold reduce-rules-def*) *auto*

lemma *reduce-rules-mono*: *P* ⊆ *P'* ⟹ *reduce-rules δ P* ⊆ *reduce-rules δ P'*
by (*unfold reduce-rules-def*) *auto*

```
lemma δ-states-reduce-subset:
  shows δ-states (reduce-rules δ Q) ⊆ δ-states δ ∩ Q
  by (unfold δ-states-def reduce-rules-def)
    auto

lemmas δ-states-reduce-subsetI = rev-subsetD[OF - δ-states-reduce-subset]

definition ta-reduce
  :: ('Q,'L) tree-automaton-rec ⇒ ('Q set) ⇒ ('Q,'L) tree-automaton-rec
  where ta-reduce TA P ==
    (| ta-initial = ta-initial TA ∩ P,
      ta-rules = reduce-rules (ta-rules TA) P |)

— Reducing a tree automaton preserves the tree automata invariants
theorem ta-reduce-inv: assumes A: tree-automaton TA
  shows tree-automaton (ta-reduce TA P)
proof −
  interpret tree-automaton TA using A .
  show ?thesis proof
    show finite (ta-rules (ta-reduce TA P))
        finite (ta-initial (ta-reduce TA P))
      using finite-states finite-rules finite-subset[OF reduce-rules-subset]
      by (unfold ta-reduce-def) (auto simp add: Let-def)
  qed
qed

lemma reduce-δ-states-rules[simp]:
  (ta-rules (ta-reduce TA (δ-states (ta-rules TA)))) = ta-rules TA
  by (auto simp add: ta-reduce-def δ-states-def reduce-rules-def)

— Reducing a tree automaton to the states that occur in its rules does not change
     its language
lemma ta-reduce-δ-states:
  ta-lang (ta-reduce TA (δ-states (ta-rules TA))) = ta-lang TA
  apply (auto simp add: ta-lang-def)
  apply (frule δ-states-accsI)
  apply (auto simp add: ta-reduce-def δ-states-def reduce-rules-def) [1]
  apply (frule δ-states-accsI)
  apply (auto simp add: ta-reduce-def δ-states-def reduce-rules-def) [1]
done
```

Forward Reduction We characterize the set of forward accessible states by the reflexive, transitive closure of a forward-successor (*f-succ* $\subseteq Q \times Q$) relation applied to the initial states.

The forward-successors of a state q are those states q' such that there is a rule $q \leftarrow f(\ldots q' \ldots)$.

— Forward successors
inductive-set *f-succ* **for** δ **where**
$[\![(q \to l\ qs) \in \delta;\ q' \in set\ qs]\!] \Longrightarrow (q,q') \in f\text{-}succ\ \delta$

— Alternative characterization of forward successors
lemma *f-succ-alt*: $f\text{-}succ\ \delta = \{(q,q').\ \exists l\ qs.\ (q \to l\ qs) \in \delta \wedge q' \in set\ qs\}$
by (*auto intro*: *f-succ.intros elim*!: *f-succ.cases*)

— Forward accessible states
definition $f\text{-}accessible\ \delta\ Q0 == ((f\text{-}succ\ \delta)^*)\ ``\ Q0$

— Alternative characterization of forward accessible states. The initial states are forward accessible, and if there is a rule whose lhs-state is forward-accessible, all rhs-states of that rule are forward-accessible, too.
inductive-set $f\text{-}accessible\text{-}alt :: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow 'Q\ set \Rightarrow 'Q\ set$
for $\delta\ Q0$
where
fa-refl: $q0 \in Q0 \Longrightarrow q0 \in f\text{-}accessible\text{-}alt\ \delta\ Q0$ |
fa-step: $[\![\ q \in f\text{-}accessible\text{-}alt\ \delta\ Q0;\ (q \to l\ qs) \in \delta;\ q' \in set\ qs\ ]\!]$
$\Longrightarrow q' \in f\text{-}accessible\text{-}alt\ \delta\ Q0$

lemma *f-accessible-alt*: $f\text{-}accessible\ \delta\ Q0 = f\text{-}accessible\text{-}alt\ \delta\ Q0$
apply (*unfold f-accessible-def f-succ-alt*)
apply *auto*
proof *goal-cases*
case *1* **thus** *?case*
apply (*induct rule*: *rtrancl-induct*)
apply (*auto intro*: *f-accessible-alt.intros*)
done
next
case *2* **thus** *?case*
apply (*induct rule*: *f-accessible-alt.induct*)
apply (*auto simp add*: *Image-def intro*: *rtrancl.intros*)
done
qed

lemmas *f-accessibleI* = *f-accessible-alt.intros*[*folded f-accessible-alt*]
lemmas *f-accessibleE* = *f-accessible-alt.cases*[*folded f-accessible-alt*]

lemma *f-succ-finite*[*simp*, *intro*]: *finite* $\delta \Longrightarrow$ *finite* (*f-succ* δ)
apply (*unfold f-succ-alt*)
apply (*rule-tac* $B{=}\delta\text{-}states\ \delta \times \delta\text{-}states\ \delta$ **in** *finite-subset*)
apply (*auto dest*: $\delta\text{-}statesI$ *simp add*: $\delta\text{-}states\text{-}finite$)
done

lemma *f-accessible-mono*: $Q \subseteq Q' \Longrightarrow x \in f\text{-}accessible\ \delta\ Q \Longrightarrow x \in f\text{-}accessible\ \delta\ Q'$
by (*auto simp add*: *f-accessible-def*)

lemma *f-accessible-prepend*:

```
  ⟦ (q → l qs) ∈ δ; q'∈set qs; x∈f-accessible δ {q'} ⟧
    ⟹ x∈f-accessible δ {q}
  by (auto dest: f-succ.intros simp add: f-accessible-def)

lemma f-accessible-subset: q∈f-accessible δ Q ⟹ q∈Q ∪ δ-states δ
  apply (unfold f-accessible-alt)
  apply (induct rule: f-accessible-alt.induct)
  apply (force simp add: δ-states-def split: ta-rule.split-asm)+
  done

lemma (in tree-automaton) f-accessible-in-states:
  q∈f-accessible (ta-rules TA) (ta-initial TA) ⟹ q∈ta-rstates TA
  using initial-subset states-subset
  by (drule-tac f-accessible-subset) (auto)

lemma f-accessible-refl-inter-simp[simp]: Q ∩ f-accessible r Q = Q
  by (unfold f-accessible-alt) (auto intro: fa-refl)

  — A tree remains accepted by a state q if the rules are reduced to the states that
      are forward-accessible from q
lemma accs-reduce-f-acc:
  accs δ t q ⟹ accs (reduce-rules δ (f-accessible δ {q})) t q
proof (induct rule: accs.induct[case-names step])
  case (step q l qs δ n)
  show ?case proof (rule accs.intros[of q l qs])
    show (q → l qs) ∈ reduce-rules δ (f-accessible δ {q})
      using step(1)
      by (fastforce
        intro!: reduce-rulesI
        intro: f-succ.intros
        simp add: f-accessible-def)
  next
    fix i
    assume A: i<length qs

    have B: f-accessible δ {q} ⊇ f-accessible δ {qs!i} using step.hyps(1)
      by (force
        simp add: A f-accessible-def
        intro: converse-rtrancl-into-rtrancl f-succ.intros[where q'=qs!i])
    show accs (reduce-rules δ (f-accessible δ {q})) (n ! i) (qs ! i)
      using accs-mono[OF step.hyps(4)[OF A] reduce-rules-mono[OF B]] .
  qed (simp-all add: step.hyps(2,3))
qed

  — Short-hand notation for forward-reducing a tree-automaton
abbreviation ta-fwd-reduce TA ==
  (ta-reduce TA (f-accessible (ta-rules TA) (ta-initial TA)))

— Forward-reducing a tree automaton does not change its language
```

theorem *ta-reduce-f-acc*[*simp*]: *ta-lang* (*ta-fwd-reduce TA*) = *ta-lang TA*
apply (*rule sym*)
apply (*unfold ta-reduce-def ta-lang-def*)
apply (*auto simp add*: *Let-def*)
apply (*rule-tac x*=*q* **in** *bexI*)
apply (*drule accs-reduce-f-acc*)
apply (*rule-tac*
P1=(*f-accessible* (*ta-rules TA*) {*q*})
in *accs-mono*[*OF - reduce-rules-mono*])
apply (*auto simp add*: *f-accessible-def*)
apply (*rule-tac x*=*q* **in** *bexI*)
apply (*blast intro*: *accs-mono*[*OF - reduce-rules-subset*])
.

Backward Reduction A state is backward accessible, iff at least one tree is accepted in it.

Inductively, backward accessible states can be characterized as follows: A state is backward accessible, if it occurs on the left hand side of a rule, and all states on this rule's right hand side are backward accessible.

inductive-set *b-accessible* :: ('*Q*,'*L*) *ta-rule set* ⇒ '*Q set*
for *δ*
where
⟦ (*q* → *l qs*)∈*δ*; !!*x*. *x*∈*set qs* ⟹ *x*∈*b-accessible δ* ⟧ ⟹ *q*∈*b-accessible δ*

lemma *b-accessibleI*:
⟦(*q* → *l qs*)∈*δ*; *set qs* ⊆ *b-accessible δ*⟧ ⟹ *q*∈*b-accessible δ*
by (*auto intro*: *b-accessible.intros*)

— States that accept a tree are backward accessible
lemma *accs-is-b-accessible*: *accs δ t q* ⟹ *q*∈*b-accessible δ*
apply (*induct rule*: *accs.induct*)
apply (*rule b-accessible.intros*)
apply *assumption*
apply (*fastforce simp add*: *in-set-conv-nth*)
done

lemma *b-acc-subset-δ-statesI*: *x*∈*b-accessible δ* ⟹ *x*∈*δ-states δ*
apply (*erule b-accessible.cases*)
apply (*auto intro*: *δ-statesI*)
done

lemma *b-acc-subset-δ-states*: *b-accessible δ* ⊆ *δ-states δ*
by (*auto simp add*: *b-acc-subset-δ-statesI*)

lemma *b-acc-finite*[*simp*, *intro*!]: *finite δ* ⟹ *finite* (*b-accessible δ*)
apply (*rule finite-subset*[*OF b-acc-subset-δ-states*])
apply *auto*

```
  done

  — Backward accessible states accept at least one tree
lemma b-accessible-is-accs:
  ⟦ q∈b-accessible (ta-rules TA);
    !!t. accs (ta-rules TA) t q ⟹ P
  ⟧ ⟹ P
proof (induct arbitrary: P rule: b-accessible.induct[case-names IH])
  case (IH q l qs)

  obtain ts where
    A: ∀ i<length qs. accs (ta-rules TA) (ts!i) (qs!i)
       length ts = length qs
  proof −
    from IH(3) have ∀ x∈set qs. ∃ t. accs (ta-rules TA) t x by auto
    hence ∃ ts. (∀ i<length qs. accs (ta-rules TA) (ts!i) (qs!i))
              ∧ length ts = length qs
    proof (induct qs)
      case Nil thus ?case by simp
    next
      case (Cons q qs) then obtain ts where
        IHAPP: ∀ i<length qs. accs (ta-rules TA) (ts ! i) (qs ! i) and
        L: length ts = length qs
        by auto
      moreover from Cons obtain t where accs (ta-rules TA) t q by auto
      ultimately have
        ∀ i<length (q#qs). accs (ta-rules TA) ((t#ts) ! i) ((q#qs) ! i)
        apply auto
        apply (case-tac i)
        apply auto
        done
      thus ?case using L by auto
    qed
    thus thesis by (blast intro: that)
  qed

  from A show ?case
    apply (rule-tac IH(4)[OF accs.intros[OF IH(1)]])
    apply auto
    done
qed

  — All trees remain accepted when reducing the rules to backward-accessible states
lemma accs-reduce-b-acc:
  accs δ t q ⟹ accs (reduce-rules δ (b-accessible δ)) t q
  apply (induct rule: accs.induct)
  apply (rule accs.intros)
  apply (rule reduce-rulesI)
  apply assumption
```

```
  apply (auto)
  apply (rule-tac t=NODE f ts in accs-is-b-accessible)
  apply (rule-tac accs.intros)
  apply auto
  apply (simp only: in-set-conv-nth)
  apply (erule-tac exE)
  apply (rule-tac t=ts ! i in accs-is-b-accessible)
  apply auto
  done

  — Shorthand notation for backward-reduction of a tree automaton
abbreviation ta-bwd-reduce TA == (ta-reduce TA (b-accessible (ta-rules TA)))

— Backwards-reducing a tree automaton does not change its language
theorem ta-reduce-b-acc[simp]: ta-lang (ta-bwd-reduce TA) = ta-lang TA
  apply (rule sym)
  apply (unfold ta-reduce-def ta-lang-def)
  apply (auto simp add: Let-def)
  apply (rule-tac x=q in bexI)
  apply (blast intro: accs-reduce-b-acc)
  apply (blast dest: accs-is-b-accessible)
  apply (rule-tac x=q in bexI)
  apply (blast intro: accs-mono[OF - reduce-rules-subset])
  .

  — Emptiness check by backward reduction. The language of a tree automaton is
      empty, if and only if no initial state is backwards-accessible.
theorem empty-if-no-b-accessible:
  ta-lang TA = {} ⟷ ta-initial TA ∩ b-accessible (ta-rules TA) = {}
  by (auto
    simp add: ta-lang-def
    intro: accs-is-b-accessible b-accessible-is-accs)
```

3.4.5 Product Automaton

The product automaton of two tree automata accepts the intersection of the languages of the two automata.

```
  — Product rule
fun r-prod where
  r-prod (q1 → l1 qs1) (q2 → l2 qs2) = ((q1,q2) → l1 (zip qs1 qs2))

  — Product rules
definition δ-prod δ1 δ2 == {
  r-prod (q1 → l qs1) (q2 → l qs2) | q1 q2 l qs1 qs2.
    length qs1 = length qs2 ∧
    (q1 → l qs1)∈δ1 ∧
    (q2 → l qs2)∈δ2
}
```

```
lemma δ-prodI: ⟦
    length qs1 = length qs2;
    (q1 → l qs1)∈δ1;
    (q2 → l qs2)∈δ2 ⟧ ⟹ ((q1,q2) → l (zip qs1 qs2)) ∈ δ-prod δ1 δ2
  by (auto simp add: δ-prod-def)

lemma δ-prodE:
  ⟦
    r∈δ-prod δ1 δ2;
    !!q1 q2 l qs1 qs2. ⟦ length qs1 = length qs2;
                         (q1 → l qs1)∈δ1;
                         (q2 → l qs2)∈δ2;
                         r = ((q1,q2) → l (zip qs1 qs2))
                       ⟧ ⟹ P
  ⟧ ⟹ P
  by (auto simp add: δ-prod-def)

  — With the product rules, only trees can be constructed that can also be con-
      structed with the two original sets of rules
lemma δ-prod-sound:
  assumes A: accs (δ-prod δ1 δ2) t (q1,q2)
  shows accs δ1 t q1    accs δ2 t q2
proof −
  {
    fix δ q
    assume accs δ t q    δ = (δ-prod δ1 δ2)    q=(q1,q2)
    hence accs δ1 t q1 ∧ accs δ2 t q2
      by (induct arbitrary: δ1 δ2 q1 q2 rule: accs.induct)
        (auto intro: accs.intros simp add: δ-prod-def)
  } with A show accs δ1 t q1    accs δ2 t q2 by auto
qed

  — Any tree that can be constructed with both original sets of rules can also be
      constructed with the product rules
lemma δ-prod-precise:
  ⟦ accs δ1 t q1; accs δ2 t q2 ⟧ ⟹ accs (δ-prod δ1 δ2) t (q1,q2)
proof (induct arbitrary: δ2 q2 rule: accs.induct[case-names step])
  case (step q1 l qs1 δ1 ts δ2 q2)
  note [simp] = step.hyps(2,3)
  from step.hyps(2) obtain qs2 where
    I2: (q2 → l qs2)∈δ2
        !!i. i<length qs2 ⟹ accs δ2 (ts ! i) (qs2 ! i) and
    [simp]: length qs2 = length ts
    by (rule-tac accs.cases[OF step.prems]) fastforce
  show ?case
  proof (rule accs.intros)
    from step.hyps(1) I2(1) show
      ((q1,q2) → l (zip qs1 qs2))∈δ-prod δ1 δ2 and
      [simp]: length ts = length (zip qs1 qs2)
```

```
    by (unfold δ-prod-def) force+
  next
    fix i
    assume L: i<length (zip qs1 qs2)
    with step.hyps(4)[OF - I2(2), of i] have
      accs (δ-prod δ1 δ2) (ts ! i) (qs1 ! i, qs2 ! i)
      by simp
    also have (qs1 ! i, qs2 ! i) = zip qs1 qs2 ! i using L by auto
    finally show accs (δ-prod δ1 δ2) (ts ! i) (zip qs1 qs2 ! i) .
  qed
qed

lemma δ-prod-empty[simp]:
  δ-prod {} δ = {}
  δ-prod δ {} = {}
  by (unfold δ-prod-def) auto

lemma δ-prod-2sng[simp]:
  ⟦ rhsl r1 ≠ rhsl r2 ⟧ ⟹ δ-prod {r1} {r2} = {}
  ⟦ length (rhsq r1) ≠ length (rhsq r2) ⟧ ⟹ δ-prod {r1} {r2} = {}
  ⟦ rhsl r1 = rhsl r2; length (rhsq r1) = length (rhsq r2) ⟧
    ⟹ δ-prod {r1} {r2} = {r-prod r1 r2}
  apply (unfold δ-prod-def)
  apply (cases r1, cases r2, auto)+
  done

lemma δ-prod-Un[simp]:
  δ-prod (δ1∪δ1') δ2 = δ-prod δ1 δ2 ∪ δ-prod δ1' δ2
  δ-prod δ1 (δ2∪δ2') = δ-prod δ1 δ2 ∪ δ-prod δ1 δ2'
  by (auto elim: δ-prodE intro: δ-prodI)
```

The next two definitions are solely for technical reasons. They are required to allow simplification of expressions of the form *δ-prod (insert r δ1) δ2* or *δ-prod δ1 (insert r δ2)*, without making the simplifier loop.

```
definition δ-prod-sng1 r δ2 ==
  case r of (q1 → l qs1) ⇒
    { r-prod r (q2 → l qs2) |
        q2 qs2. length qs1 = length qs2 ∧ (q2 → l qs2)∈δ2
    }
definition δ-prod-sng2 δ1 r ==
  case r of (q2 → l qs2) ⇒
    { r-prod (q1 → l qs1) r |
        q1 qs1. length qs1 = length qs2 ∧ (q1 → l qs1)∈δ1
    }

lemma δ-prod-sng-alt:
  δ-prod-sng1 r δ2 = δ-prod {r} δ2
  δ-prod-sng2 δ1 r = δ-prod δ1 {r}
  apply (unfold δ-prod-def δ-prod-sng1-def δ-prod-sng2-def)
```

```
  apply (auto split: ta-rule.split)
  done

lemmas δ-prod-insert =
  δ-prod-Un(1)[where ?δ1.0={x}, simplified, folded δ-prod-sng-alt]
  δ-prod-Un(2)[where ?δ2.0={x}, simplified, folded δ-prod-sng-alt]
  for x

  — Product automaton
definition ta-prod TA1 TA2 ==
  (| ta-initial = ta-initial TA1 × ta-initial TA2,
    ta-rules = δ-prod (ta-rules TA1) (ta-rules TA2)
  |)

lemma ta-prod-correct-aux1:
  ta-lang (ta-prod TA1 TA2) = ta-lang TA1 ∩ ta-lang TA2
  by (unfold ta-lang-def ta-prod-def) (auto dest: δ-prod-sound δ-prod-precise)

lemma δ-states-cart:
  q ∈ δ-states (δ-prod δ1 δ2) ⟹ q ∈ δ-states δ1 × δ-states δ2
  by (unfold δ-states-def δ-prod-def)
     (force split: ta-rule.split simp add: set-zip)

lemma δ-prod-finite [simp, intro]:
  finite δ1 ⟹ finite δ2 ⟹ finite (δ-prod δ1 δ2)
proof −
  have
    δ-prod δ1 δ2
    ⊆ (λ(r1,r2). case r1 of (q1 → l1 qs1) ⇒
                 case r2 of (q2 → l2 qs2) ⇒
                   ((q1,q2) → l1 (zip qs1 qs2)))
       ` (δ1 × δ2)
    by (unfold δ-prod-def) force
  moreover assume finite δ1    finite δ2
  ultimately show ?thesis
    by (metis finite-imageI finite-cartesian-product finite-subset)
qed

lemma ta-prod-correct-aux2:
  assumes TA: tree-automaton TA1    tree-automaton TA2
  shows tree-automaton (ta-prod TA1 TA2)
proof −
  interpret ta1: tree-automaton TA1 using TA by blast
  interpret ta2: tree-automaton TA2 using TA by blast
  show ?thesis
    apply unfold-locales
    apply (unfold ta-prod-def)
    apply (auto
      intro: ta1.is-subset ta2.is-subset δ-prod-finite
```

dest: *δ-states-cart*
simp add: *ta1.finite-states ta2.finite-states*
ta1.finite-rules ta2.finite-rules)
done
qed

— The language of the product automaton is the intersection of the languages of the two original automata
theorem *ta-prod-correct*:
assumes *TA*: *tree-automaton TA1* *tree-automaton TA2*
shows
ta-lang (*ta-prod TA1 TA2*) = *ta-lang TA1* ∩ *ta-lang TA2*
tree-automaton (*ta-prod TA1 TA2*)
using *ta-prod-correct-aux1*
ta-prod-correct-aux2[*OF TA*]
by *auto*

lemma *ta-prod-rta*:
assumes *TA*: *ranked-tree-automaton TA1 A* *ranked-tree-automaton TA2 A*
shows *ranked-tree-automaton* (*ta-prod TA1 TA2*) *A*
proof −
interpret *ta1*: *ranked-tree-automaton TA1 A* **using** *TA* **by** *blast*
interpret *ta2*: *ranked-tree-automaton TA2 A* **using** *TA* **by** *blast*

interpret *tap*: *tree-automaton* (*ta-prod TA1 TA2*)
apply (*rule ta-prod-correct-aux2*)
by *unfold-locales*

show *?thesis*
apply *unfold-locales*
apply (*unfold ta-prod-def δ-prod-def*)
apply (*auto intro*: *ta1.ranked ta2.ranked*)
done
qed

3.4.6 Determinization

We only formalize the brute-force subset construction without reduction.

The basic idea of this construction is to construct an automaton where the states are sets of original states, and the lhs of a rule consists of all states that a term with given rhs and function symbol may be labeled by.

context *ranked-tree-automaton*
begin
— Left-hand side of subset rule for given symbol and rhs
definition *δss-lhs f ss* ==
{ *q* | *q qs*. (*q* → *f qs*)∈*δ* ∧ *list-all-zip* (∈) *qs ss* }

— Subset construction
inductive-set $\delta ss :: ('Q\ set, 'L)\ ta\text{-}rule\ set$ **where**
  $[\![\ A\ f = Some\ (length\ ss);$
    $ss \in lists\ \{s.\ s \subseteq ta\text{-}rstates\ TA\};$
    $s = \delta ss\text{-}lhs\ f\ ss$
  $]\!] \Longrightarrow (s \to f\ ss) \in \delta ss$

lemma δssI:
  assumes A: $A\ f = Some\ (length\ ss)$
      $ss \in lists\ \{s.\ s \subseteq ta\text{-}rstates\ TA\}$
  shows
    $(\ (\delta ss\text{-}lhs\ f\ ss) \to f\ ss) \in \delta ss$
  using $\delta ss.intros$[**where** s=$(\delta ss\text{-}lhs\ f\ ss)$] A
  by *auto*

lemma $\delta ss\text{-}subset$[*simp*, *intro*!]: $\delta ss\text{-}lhs\ f\ ss \subseteq Q$
  by (*unfold ta-rstates-def* $\delta ss\text{-}lhs\text{-}def$) (*auto intro*: $\delta\text{-}statesI$)

lemma $\delta ss\text{-}finite$[*simp*, *intro*!]: *finite* δss
proof −
  have $\delta ss \subseteq \bigcup((\lambda f.\ (\lambda(s,ss).\ (s \to f\ ss))$
        ‘$(\{s.\ s \subseteq Q\}$
          $\times\ (lists\ \{s.\ s \subseteq Q\} \cap \{l.\ length\ l = the\ (A\ f)\})$)
) ‘ F)
    (**is** -$\subseteq\bigcup((\lambda f.\ ?tr\ f$ ‘ $?prod\ f)$‘F))
  proof (*intro equalityI subsetI*)
    fix r
    assume $r \in \delta ss$
    then obtain $f\ s\ ss$ **where**
      U: r=$(s \to f\ ss)$
        $A\ f = Some\ (length\ ss)$
        ss∈$lists\ \{s.\ s \subseteq Q\}$
        s=$\delta ss\text{-}lhs\ f\ ss$
      by (*force elim*!: $\delta ss.cases$)
    from $U(4)$ **have** $s \subseteq Q$ **by** *simp*
    moreover from $U(2)$ **have** $length\ ss = the\ (A\ f)$ **by** *simp*
    ultimately have $(s,ss) \in ?prod\ f$ **using** $U(3)$ **by** *auto*
    hence $(s \to f\ ss) \in ?tr\ f$ ‘ $?prod\ f$ **by** *auto*
    moreover from $U(2)$ **have** $f \in F$ **by** *auto*
    ultimately show $r \in \bigcup((\lambda f.\ ?tr\ f$ ‘ $?prod\ f)$‘$F)$ **using** $U(1)$ **by** *auto*
  qed
  moreover have *finite* ...
    by (*auto intro*!: *finite-imageI finite-SigmaI lists-of-len-fin*)
  ultimately show *?thesis* **by** (*blast intro*: *finite-subset*)
qed

lemma $\delta ss\text{-}det$: $[\![\ (q \to f\ qs) \in \delta ss;\ (q' \to f\ qs) \in \delta ss\ ]\!] \Longrightarrow q$=$q'$
  by (*auto elim*!: $\delta ss.cases$)

```
lemma δss-accs-sound:
  assumes A: accs δ t q
  obtains s where
  s⊆Q
  q∈s
  accs δss t s
proof −
  have ∃ s⊆Q. q∈s ∧ accs-laz δss t s using A[unfolded accs-laz]
  proof (induct δ≡δ t q rule: accs-laz.induct[case-names step])
    case (step q f qs ts)
    hence I:
      (q → f qs)∈δ
      list-all-zip (accs-laz δ) ts qs
      list-all-zip (λt q. ∃ s. s⊆Q ∧ q∈s ∧ accs-laz δss t s) ts qs
      by simp-all
    from I(3) obtain ss where SS:
      ss ∈ lists {s. s⊆Q}
      list-all-zip (∈) qs ss
      list-all-zip (accs-laz δss) ts ss
      by (erule-tac laz-swap-ex) auto
    from I(2) SS(2) have
      LEN[simp]: length qs = length ts    length ss = length ts
      by (auto simp add: list-all-zip-alt)
    from ranked[OF I(1)] have AF: A f = Some (length ts) by simp

    from δssI[of f ss, OF - SS(1)] AF have
      RULE-S: ((δss-lhs f ss) → f ss) ∈ δss
      by simp

    from accs-laz.intros[OF RULE-S SS(3)] have
      G1: accs-laz δss (NODE f ts) (δss-lhs f ss) .
    from I(1) SS(2) have q∈(δss-lhs f ss) by (auto simp add: δss-lhs-def)
    thus ?case using G1 by auto
  qed
  thus ?thesis
    apply (elim exE conjE)
    apply (rule-tac that)
    apply assumption
    apply (auto simp add: accs-laz)
    done
qed


lemma δss-accs-precise:
  assumes A: accs δss t s    q∈s
  shows accs δ t q
  using A
  unfolding accs-laz
proof (induct δ≡δss t s
```

```
        arbitrary: q
        rule: accs-laz.induct[case-names step])
  case (step s f ss ts)
  hence I:
    (s → f ss) ∈ δss
    list-all-zip (accs-laz δss) ts ss
    list-all-zip (λt s. ∀ q∈s. accs-laz δ t q) ts ss
    q∈s
    by (auto simp add: Ball-def)

  from I(2) have [simp]: length ss = length ts
    by (simp add: list-all-zip-alt)

  from I(1) have SS:
    A f = Some (length ts)
    ss ∈ lists {s. s⊆Q}
    s=δss-lhs f ss
    by (force elim!: δss.cases)+

  from I(4) SS(3) obtain qs where
    RULE: (q → f qs) ∈ δ and
    QSISS: list-all-zip (∈) qs ss
    by (auto simp add: δss-lhs-def)
  from I(3) QSISS have CA: list-all-zip (accs-laz δ) ts qs
    by (auto simp add: list-all-zip-alt)
  from accs-laz.intros[OF RULE CA] show ?case .
qed


— Determinization
definition detTA == (| ta-initial = { s. s⊆Q ∧ s∩Qi ≠ {} },
                       ta-rules = δss |)

theorem detTA-is-ta[simp, intro]:
  det-tree-automaton detTA A
  apply (unfold-locales)
  apply (auto simp add: detTA-def elim: δss.cases)
  done


theorem detTA-lang[simp]:
  ta-lang (detTA) = ta-lang TA
  apply (unfold ta-lang-def detTA-def)
  apply safe
  apply simp-all
proof −
  fix t s
  assume A:
    s⊆Q ∧ s∩Qi ≠ {}
```

```
    accs δss t s
  from A(1) obtain qi where QI: qi∈s    qi∈Qi by auto

  from δss-accs-precise[OF A(2) QI(1)] have accs δ t qi .
  with QI(2) show ∃ qi∈Qi. accs δ t qi by blast
 next
  fix t qi
  assume A:
    qi∈Qi
    accs δ t qi
  from δss-accs-sound[OF A(2)] obtain s where SS:
    s⊆Q
    qi∈s
    accs δss t s .
  with A(1) show ∃ s⊆Q. s ∩ Qi ≠ {} ∧ accs δss t s by blast
 qed

 lemmas detTA-correct = detTA-is-ta detTA-lang
end
```

3.4.7 Completion

To each deterministic tree automaton, rules and states can be added to make it complete, without changing its language.

```
context det-tree-automaton
begin
 — States of the complete automaton
 definition Qcomplete == insert None (Some‘Q)

 lemma Qcomplete-finite[simp, intro!]: finite Qcomplete
  by (auto simp add: Qcomplete-def)

 — Rules of the complete automaton
 definition δcomplete :: ('Q option, 'L) ta-rule set where
  δcomplete == (remap-rule Some ‘ δ)
               ∪ { (None → f qs) | f qs.
                     A f = Some (length qs)
                     ∧ qs∈lists Qcomplete
                     ∧ ¬(∃ qo qso. (qo → f qso)∈δ ∧ qs=map Some qso ) }


 lemma δ-states-complete: q∈δ-states δcomplete ⟹ q∈Qcomplete
  apply (erule δ-statesE)
  apply (unfold δcomplete-def Qcomplete-def)
  apply auto
  apply (case-tac x)
  apply (auto simp add: ta-rstates-def intro: δ-statesI) [1]
  apply (case-tac x)
  apply (auto simp add: ta-rstates-def dest: δ-statesI)
```

done

definition
completeTA == (| *ta-initial* = *Some*‘*Qi*, *ta-rules* = *δcomplete* |)

lemma *δcomplete-finite*[*simp*, *intro*]: *finite δcomplete*
proof −
have *δcomplete* ⊆ *legal-rules Qcomplete*
apply (*rule*)
apply (*rule legal-rulesI*)
apply *assumption*
apply (*case-tac x*)
apply (*unfold δcomplete-def Qcomplete-def ta-rstates-def*) [*1*]
apply *auto*
apply (*case-tac xa*)
apply (*auto dest*: *δ-statesI*)
apply (*case-tac xa*)
apply (*auto dest*: *δ-statesI*)
apply (*unfold δcomplete-def Qcomplete-def ta-rstates-def*) [*1*]
apply (*auto*)
apply (*case-tac xa*)
apply (*auto intro*: *ranked*)
done
thus *?thesis* **by** (*auto intro*: *finite-subset*)
qed

theorem *completeTA-is-ta*: *complete-tree-automaton completeTA A*
proof (*standard*, *goal-cases*)
case *1* **thus** *?case* **by** (*simp add*: *completeTA-def*)
next
case *2* **thus** *?case* **by** (*simp add*: *completeTA-def*)
next
case *3* **thus** *?case*
apply (*auto simp add*: *completeTA-def δcomplete-def*)
apply (*case-tac x*)
apply (*auto intro*: *ranked*)
done
next
case *4* **thus** *?case*
apply (*auto simp add*: *completeTA-def δcomplete-def*)
apply (*case-tac x*, *case-tac xa*)
apply (*auto intro*: *deterministic*) [*1*]
apply (*case-tac x*)
apply *auto* [*1*]
apply (*case-tac x*)
apply *auto* [*1*]
done
next

```
  case prems: (5 qs f)
  {
    fix qo qso
    assume R: (qo → f qso)∈δ and [simp]: qs=map Some qso
    hence ((Some qo) → f qs) ∈ remap-rule Some ‘ δ by force
    hence ?case by (simp add: completeTA-def δcomplete-def) blast
  } moreover {
    assume A: ¬(∃ qo qso. (qo → f qso)∈δ ∧ qs=map Some qso)

    have (Some ‘ Qi ∪ δ-states δcomplete) ⊆ Qcomplete
      apply (auto intro: δ-states-complete)
      apply (simp add: Qcomplete-def ta-rstates-def)
      done

    with prems have B: qs∈lists Qcomplete
      by (auto simp add: completeTA-def ta-rstates-def)

    from A B prems(2) have ?case
      apply (rule-tac x=None in exI)
      apply (simp add: completeTA-def δcomplete-def)
      done
  } ultimately show ?case by blast
qed

theorem completeTA-lang: ta-lang completeTA = ta-lang TA
proof (intro equalityI subsetI)
  — This direction is done by a monotonicity argument
  fix t
  assume t∈ta-lang TA
  then obtain qi where qi∈Qi    accs δ t qi by (auto simp add: ta-lang-def)
  hence
    QI: Some qi ∈ Some‘Qi and
    ACCS: accs (remap-rule Some‘δ) t (Some qi)
    by (auto intro: remap-accs1)
  have (remap-rule Some‘δ) ⊆ δcomplete by (unfold δcomplete-def) auto
  with ACCS have accs δcomplete t (Some qi) by (blast dest: accs-mono)
  thus t∈ta-lang completeTA using QI
    by (auto simp add: ta-lang-def completeTA-def)
next
  fix t
  assume A: t∈ta-lang completeTA
  then obtain qi where
    QI: qi∈Qi and
    ACCS: accs δcomplete t (Some qi)
    by (auto simp add: ta-lang-def completeTA-def)
  moreover
  {
    fix qi
    have ⟦ accs δcomplete t (Some qi) ⟧ ⟹ accs δ t qi
```

```
    unfolding accs-laz
   proof (induct δ≡δcomplete t q≡Some qi
            arbitrary: qi
            rule: accs-laz.induct[case-names step])
    case (step f qs ts qi)
    hence I:
     ((Some qi) → f qs) ∈ δcomplete
     list-all-zip (accs-laz δcomplete) ts qs
     list-all-zip (λt q. (∀ qo. q=Some qo ⟶ accs-laz δ t qo)) ts qs
     by auto
    from I(1) have ((Some qi) → f qs) ∈ remap-rule Some‘δ
     by (unfold δcomplete-def) auto
    then obtain qso where
     RULE: (qi → f qso)∈δ and
     QSF: qs=map Some qso
     apply (auto)
     apply (case-tac x)
     apply auto
     done
    from I(3) QSF have ACCS: list-all-zip (accs-laz δ) ts qso
     by (auto simp add: list-all-zip-alt)
    from accs-laz.intros[OF RULE ACCS] show ?case .
   qed
  }
  ultimately have accs δ t qi by blast
  thus t∈ta-lang TA using QI by (auto simp add: ta-lang-def)
 qed

 lemmas completeTA-correct = completeTA-is-ta completeTA-lang
end
```

3.4.8 Complement

A deterministic, complete tree automaton can be transformed into an automaton accepting the complement language by complementing its initial states.

```
context complete-tree-automaton
begin

  — Complement automaton, i.e. that accepts exactly the trees not accepted by
    this automaton
 definition complementTA == (|
  ta-initial = Q − Qi,
  ta-rules = δ |)


 lemma cta-rules[simp]: ta-rules complementTA = δ
  by (auto simp add: complementTA-def)
```

```
  theorem complementTA-correct:
    ta-lang complementTA = ranked-trees A − ta-lang TA (is ?T1)
    complete-tree-automaton complementTA A (is ?T2)
  proof −
    show ?T1
      apply (unfold ta-lang-def complementTA-def)
      apply (force intro: accs-is-ranked dest: accs-unique label-all)
      done

    have QSS: !!q. q∈ta-rstates complementTA ⟹ q∈Q
      by (auto simp add: complementTA-def ta-rstates-def)

    show ?T2
      apply (unfold-locales)
      apply (unfold complementTA-def)[4]
      apply (auto simp add: deterministic ranked
                  intro: complete QSS)
      done
  qed

end
```

3.5 Regular Tree Languages

3.5.1 Definitions

```
  — Regular languages over alphabet A
definition regular-languages :: ('L ⇀ nat) ⇒ 'L tree set set
  where regular-languages A ==
    { ta-lang TA | (TA::(nat,'L) tree-automaton-rec).
                        ranked-tree-automaton TA A }


lemma rtlE:
  fixes L :: 'L tree set
  assumes A: L∈regular-languages A
  obtains TA::(nat,'L) tree-automaton-rec where
    L=ta-lang TA
    ranked-tree-automaton TA A
  using A that
  by (unfold regular-languages-def) blast

context ranked-tree-automaton
begin

  lemma (in ranked-tree-automaton) rtlI[simp]:
    shows ta-lang TA ∈ regular-languages A
  proof −
    — Obtain injective mapping from the finite set of states to the natural numbers
    from finite-imp-inj-to-nat-seg[OF finite-states] obtain f :: 'Q ⇒ nat
```

```
    where INJMAP: inj-on f (ta-rstates TA) by blast
  — Remap automaton. The language remains the same.
  from remap-lang[OF INJMAP] have LE: ta-lang (ta-remap f TA) = ta-lang
TA .
  moreover have ranked-tree-automaton (ta-remap f TA) A ..
  ultimately show ?thesis by (auto simp add: regular-languages-def)
 qed
```

It is sometimes more handy to obtain a complete, deterministic tree automaton accepting a given regular language.

```
 theorem obtain-complete:
  obtains TAC::('Q set option,'L) tree-automaton-rec where
  ta-lang TAC = ta-lang TA
  complete-tree-automaton TAC A
 proof −
  from detTA-correct have
    DT: det-tree-automaton detTA A and
    [simp]: ta-lang detTA = ta-lang TA
    by simp-all

  interpret dt: det-tree-automaton detTA A using DT .

  from dt.completeTA-correct have G:
    ta-lang (det-tree-automaton.completeTA detTA A) = ta-lang TA
    complete-tree-automaton (det-tree-automaton.completeTA detTA A) A
    by simp-all
  thus ?thesis by (blast intro: that)
 qed

end


lemma rtlE-complete:
 fixes L :: 'L tree set
 assumes A: L∈regular-languages A
 obtains TA::(nat,'L) tree-automaton-rec where
  L=ta-lang TA
  complete-tree-automaton TA A
proof −
 from rtlE[OF A] obtain TA :: (nat,'L) tree-automaton-rec where
  [simp, symmetric]: L = ta-lang TA and
     RT: ranked-tree-automaton TA A .

 interpret ta: ranked-tree-automaton TA A using RT .

 obtain TAC :: (nat set option,'L) tree-automaton-rec
  where [simp]: ta-lang TAC = L and CT: complete-tree-automaton TAC A
  by (rule-tac ta.obtain-complete) auto
```

interpret *tac*: *complete-tree-automaton TAC A* **using** *CT* **.**

— Obtain injective mapping from the finite set of states to the natural numbers
from *finite-imp-inj-to-nat-seg*[*OF tac.finite-states*]
obtain *f* :: *nat set option* ⇒ *nat* **where**
INJMAP: *inj-on f* (*ta-rstates TAC*) **by** *blast*
— Remap automaton. The language remains the same.
from *tac.remap-lang*[*OF INJMAP*] **have** *LE*: *ta-lang* (*ta-remap f TAC*) = *L*
by *simp*
have *complete-tree-automaton* (*ta-remap f TAC*) *A*
using *tac.remap-cta*[*OF INJMAP*] **.**
thus *?thesis* **by** (*rule-tac that*[*OF LE*[*symmetric*]])
qed

3.5.2 Closure Properties

In this section, we derive the standard closure properties of regular languages, i.e. that regular languages are closed under union, intersection, complement, and difference, as well as that the empty and the universal language are regular.

Note that we do not formalize homomorphisms or tree transducers here.

theorem (**in** *finite-alphabet*) *rtl-empty*[*simp*, *intro*!]: {} ∈ *regular-languages A*
by (*rule ranked-tree-automaton.rtlI*[*OF ta-empty-rta*, *simplified*])

theorem *rtl-union-closed*:
⟦ *L1*∈*regular-languages A*; *L2*∈*regular-languages A* ⟧
⟹ *L1*∪*L2* ∈ *regular-languages A*
proof (*elim rtlE*)
fix *TA1 TA2*
assume *TA*[*simp*]: *ranked-tree-automaton TA1 A* *ranked-tree-automaton TA2 A*
and [*simp*]: *L1*=*ta-lang TA1* *L2*=*ta-lang TA2*

interpret *ta1*: *ranked-tree-automaton TA1 A* **by** *simp*
interpret *ta2*: *ranked-tree-automaton TA2 A* **by** *simp*

have *ta-lang* (*ta-union-wrap TA1 TA2*) = *ta-lang TA1* ∪ *ta-lang TA2*
apply (*rule ta-union-wrap-correct*)
by *unfold-locales*
with *ranked-tree-automaton.rtlI*[*OF ta-union-wrap-rta*[*OF TA*]] **show** *?thesis*
by (*simp*)

qed

theorem *rtl-inter-closed*:
⟦*L1*∈*regular-languages A*; *L2*∈*regular-languages A*⟧ ⟹

```
    L1∩L2 ∈ regular-languages A
proof (elim rtlE, goal-cases)
  case (1 TA1 TA2)
  with ta-prod-correct[of TA1 TA2] ta-prod-rta[of TA1 A TA2] have
     L: ta-lang (ta-prod TA1 TA2) = L1∩L2  and
     A: ranked-tree-automaton (ta-prod TA1 TA2) A
    by (simp-all add: ranked-tree-automaton.axioms)
  show ?thesis using ranked-tree-automaton.rtlI[OF A]
    by (simp add: L)
qed

theorem rtl-complement-closed:
  L∈regular-languages A ⟹ ranked-trees A − L ∈ regular-languages A
proof (elim rtlE-complete, goal-cases)
  case prems: (1 TA)
  then interpret ta: complete-tree-automaton TA A by simp

  from ta.complementTA-correct have
    [simp]: ta-lang (ta.complementTA) = ranked-trees A − ta-lang TA and
    CTA: complete-tree-automaton ta.complementTA A by auto
  interpret cta: complete-tree-automaton ta.complementTA A using CTA .

  from cta.rtlI prems(1) show ?case by simp
qed

theorem (in finite-alphabet) rtl-univ:
  ranked-trees A ∈ regular-languages A
  using rtl-complement-closed[OF rtl-empty]
  by simp

theorem rtl-diff-closed:
  fixes L1 :: 'L tree set
  assumes A[simp]: L1 ∈ regular-languages A    L2∈regular-languages A
  shows L1−L2 ∈ regular-languages A
proof −
  from rtlE[OF A(1)] obtain TA1::(nat,'L) tree-automaton-rec where
    L1: L1=ta-lang TA1 and
    RT1: ranked-tree-automaton TA1 A
    .
  from rtlE[OF A(2)] obtain TA2::(nat,'L) tree-automaton-rec where
    L2: L2=ta-lang TA2 and
    RT2: ranked-tree-automaton TA2 A
    .

  interpret ta1: ranked-tree-automaton TA1 A using RT1 .
  interpret ta2: ranked-tree-automaton TA2 A using RT2 .

  from ta1.lang-is-ranked have ALT: L1−L2 = L1 ∩ (ranked-trees A − L2)
    by (auto simp add: L1 L2)
```

show *?thesis*
unfolding *ALT*
by (*simp add*: *rtl-complement-closed rtl-inter-closed*)
qed

lemmas *rtl-closed* = *finite-alphabet.rtl-empty finite-alphabet.rtl-univ*
rtl-complement-closed
rtl-inter-closed rtl-union-closed rtl-diff-closed

end

4 Abstract Tree Automata Algorithms

theory *AbsAlgo*
imports
Ta
Collections-Examples.Exploration
Collections.CollectionsV1
begin

no-notation *fun-rel-syn* (**infixr** $\rightarrow$ *60*)

This theory defines tree automata algorithms on an abstract level, that is using non-executable datatypes and constructs like sets, set-collecting operations, etc.

These algorithms are then refined to executable algorithms in Section 5.

4.1 Word Problem

First, a recursive version of the *accs*-predicate is defined.

fun *r-match* :: *'a set list* $\Rightarrow$ *'a list* $\Rightarrow$ *bool* **where**
r-match [] [] $\longleftrightarrow$ *True* |
r-match (*A*#*AS*) (*a*#*as*) $\longleftrightarrow$ *a*$\in$*A* $\wedge$ *r-match AS as* |
r-match - - $\longleftrightarrow$ *False*

— *r-match* accepts two lists, if they have the same length and the elements in the second list are contained in the respective elements of the first list:
lemma *r-match-alt*:
r-match L l $\longleftrightarrow$ *length L* = *length l* $\wedge$ ($\forall$ *i*<*length l*. *l*!*i* $\in$ *L*!*i*)
apply (*induct L l rule*: *r-match.induct*)
apply *auto*
apply (*case-tac i*)
apply *auto*
done

— Whether a rule matches the given state, label and list of sets of states
fun *r-matchc* **where**
  r-matchc q l Qs (*qr* → *lr qsr*) ⟷ *q*=*qr* ∧ *l*=*lr* ∧ *r-match Qs qsr*

— recursive version of *accs*-predicate
fun *faccs* :: ('*Q*,'*L*) *ta-rule set* ⇒ '*L tree* ⇒ '*Q set* **where**
  faccs δ (*NODE f ts*) = (
    let Qs = *map* (*faccs δ*) (*ts*) *in*
      {*q*. ∃ *r*∈*δ*. *r-matchc q f Qs r* }
)

lemma *faccs-correct-aux*:
  q∈*faccs δ n* = *accs δ n q* (**is** *?T1*)
  (*map* (*faccs δ*) *ts* = *map* (λ*t*. { *q* . *accs δ t q*}) *ts*) (**is** *?T2*)
proof −
  have (∀ *q*. *q*∈*faccs δ n* = *accs δ n q*)
       ∧ (*map* (*faccs δ*) *ts* = *map* (λ*t*. { *q* . *accs δ t q*}) *ts*)
  proof (*induct rule*: *compat-tree-tree-list.induct*)
    case (*NODE f ts*)
    thus *?case*
      apply (*intro allI iffI*)
      apply *simp*
      apply (*erule bexE*)
      apply (*case-tac x*)
      apply *simp*
      apply (*rule accs.intros*)
      apply *assumption*
      apply (*unfold r-match-alt*)
      apply *simp*
      apply *fastforce*
      apply *simp*
      apply (*erule accs.cases*)
      apply *auto*
      apply (*rule-tac x*=*qa* → *f qs* **in** *bexI*)
      apply *simp*
      apply (*unfold r-match-alt*)
      apply *auto*
      done
  qed *auto*
  thus *?T1 ?T2* **by** *auto*
qed

theorem *faccs-correct1*: *q*∈*faccs δ n* ⟹ *accs δ n q*
  by (*simp add*: *faccs-correct-aux*)
theorem *faccs-correct2*: *accs δ n q* ⟹ *q*∈*faccs δ n*
  by (*simp add*: *faccs-correct-aux*)

lemmas *faccs-correct* = *faccs-correct1 faccs-correct2*

lemma *faccs-alt*: *faccs* δ t = {q. *accs* δ t q} **by** (*auto intro*: *faccs-correct*)

4.2 Backward Reduction and Emptiness Check

4.2.1 Auxiliary Definitions

— Step function, that maps a set of states to those states that are reachable via one backward step.

inductive-set *bacc-step* :: ('Q,'L) *ta-rule set* ⇒ 'Q *set* ⇒ 'Q *set*
for δ Q
where
⟦ r∈δ; *set* (*rhsq* r) ⊆ Q ⟧ ⟹ *lhs* r ∈ *bacc-step* δ Q

— If a set is closed under adding all states that are reachable from the set by one backward step, then this set contains all backward accessible states.

lemma *b-accs-as-closed*:
assumes *A*: *bacc-step* δ Q ⊆ Q
shows *b-accessible* δ ⊆ Q
proof (*rule subsetI*)
fix q
assume q∈*b-accessible* δ
thus q∈Q
proof (*induct rule*: *b-accessible.induct*)
fix q f *qs*
assume *BC*: (q→f *qs*)∈δ
!!x. x∈*set qs* ⟹ x∈*b-accessible* δ
!!x. x∈*set qs* ⟹ x∈Q
from *bacc-step.intros*[*OF BC*(*1*)] *BC*(*3*) **have** q∈*bacc-step* δ Q **by** *auto*
with *A* **show** q∈Q **by** *blast*
qed
qed

4.2.2 Algorithms

First, the basic workset algorithm is specified. Then, it is refined to contain a counter for each rule, that counts the number of undiscovered states on the RHS. For both levels of abstraction, a version that computes the backwards reduction, and a version that checks for emptiness is specified.

Additionally, a version of the algorithm that computes a witness for non-emptiness is provided.

Levels of abstraction:

- α On this level, the state consists of a set of discovered states and a workset.
- α' On this level, the state consists of a set of discovered states, a workset and a map from rules to number of undiscovered rhs states. This map

can be used to make the discovery of rules that have to be considered more efficient.

α - Level: — A state contains the set of discovered states and a workset
type-synonym $('Q,'L)$ *br-state* $= 'Q$ *set* $\times$ $'Q$ *set*

— Set of states that are non-empty (accept a tree) after adding the state q to the set of discovered states
definition *br-dsq*
:: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $'Q \Rightarrow ('Q,'L)$ *br-state* $\Rightarrow$ $'Q$ *set*
where
br-dsq δ q $==$ $\lambda(Q,W).$ { *lhs* r | $r.$ $r{\in}\delta \wedge$ *set* (*rhsq* r) $\subseteq (Q-(W-\{q\}))$ }

— Description of a step: One state is removed from the workset, and all new states that become non-empty due to this state are added to, both, the workset and the set of discovered states
inductive-set *br-step*
:: $('Q,'L)$ *ta-rule set* $\Rightarrow ((('Q,'L)$ *br-state* $\times$ $('Q,'L)$ *br-state*$)$ *set*
for δ **where**
$[\![$
$q{\in}W$;
$Q' = Q \cup$ *br-dsq* δ q (Q,W);
$W' = W - \{q\} \cup$ (*br-dsq* δ q $(Q,W) - Q$)
$]\!] \Longrightarrow ((Q,W),(Q',W')){\in}$*br-step* δ

— Termination condition for backwards reduction: The workset is empty
definition *br-cond* :: $('Q,'L)$ *br-state set*
where *br-cond* $==$ $\{(Q,W).\ W{\neq}\{\}\}$

— Termination condition for emptiness check: The workset is empty or a non-empty initial state has been discovered
definition *bre-cond* :: $'Q$ *set* $\Rightarrow ('Q,'L)$ *br-state set*
where *bre-cond* Qi $==$ $\{(Q,W).\ W{\neq}\{\} \wedge (Qi{\cap}Q{=}\{\})\}$

— Set of all states that occur on the lhs of a constant-rule
definition *br-iq* :: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $'Q$ *set*
where *br-iq* δ $==$ { *lhs* r | $r.$ $r{\in}\delta \wedge$ *rhsq* $r = []$ }

— Initial state for the iteration
definition *br-initial* :: $('Q,'L)$ *ta-rule set* $\Rightarrow ('Q,'L)$ *br-state*
where *br-initial* δ $==$ (*br-iq* δ, *br-iq* δ)

— Invariant for the iteration:

- States on the workset have been discovered
- Only accessible states have been discovered
- If a state is non-empty due to a rule whose rhs-states have been discovered and processed (i.e. are in $Q-W$), then the lhs state of the rule has also been

discovered.

- The set of discovered states is finite

definition *br-invar* :: (′*Q*,′*L*) *ta-rule set* ⇒ (′*Q*,′*L*) *br-state set*
  where *br-invar* δ == {(*Q*,*W*).
    W⊆*Q* ∧
    Q ⊆ *b-accessible* δ ∧
    bacc-step δ (*Q* − *W*) ⊆ *Q* ∧
    finite Q}

definition *br-algo* δ == (|
  wa-cond = *br-cond*,
  wa-step = *br-step* δ,
  wa-initial = {*br-initial* δ},
  wa-invar = *br-invar* δ
|)

definition *bre-algo Qi* δ == (|
  wa-cond = *bre-cond Qi*,
  wa-step = *br-step* δ,
  wa-initial = {*br-initial* δ},
  wa-invar = *br-invar* δ
|)

— Termination: Either a new state is added, or the workset decreases.
definition *br-termrel* δ ==
  ({(*Q*′,*Q*). *Q* ⊂ *Q*′ ∧ *Q*′ ⊆ *b-accessible* δ}) <∗*lex*∗> *finite-psubset*

lemma *bre-cond-imp-br-cond*[*intro*, *simp*]: *bre-cond Qi* ⊆ *br-cond*
  by (*auto simp add*: *br-cond-def bre-cond-def*)

lemma *br-termrel-wf*[*simp*, *intro*!]: *finite* δ ⟹ *wf* (*br-termrel* δ)
  apply (*unfold br-termrel-def*)
  apply (*auto simp add*: *wf-bounded-supset*)
  done

— Only accessible states are discovered
lemma *br-dsq-ss*:
  assumes *A*: (*Q*,*W*)∈*br-invar* δ *W* ≠ {} *q*∈*W*
  shows *br-dsq* δ *q* (*Q*,*W*) ⊆ *b-accessible* δ
proof (*rule subsetI*)
  fix *q*′
  assume *B*: *q*′∈*br-dsq* δ *q* (*Q*,*W*)
  then obtain *r* **where**
    R: *q*′ = *lhs r* *r*∈δ **and**
    S: *set* (*rhsq r*) ⊆ (*Q*−(*W*−{*q*}))
    by (*unfold br-dsq-def*) *auto*
  note *S*

```
  also have (Q−(W−{q})) ⊆ b-accessible δ using A(1,3)
    by (auto simp add: br-invar-def)
  finally show q'∈b-accessible δ using R
    by (cases r)
  (auto intro: b-accessible.intros)
qed

lemma br-step-in-termrel:
  assumes A: Σ∈br-cond    Σ∈br-invar δ    (Σ,Σ')∈br-step δ
  shows (Σ', Σ)∈br-termrel δ
proof −
  obtain Q W Q' W' where
    [simp]: Σ=(Q,W)    Σ'=(Q',W')
    by (cases Σ, cases Σ', auto)
  obtain q where
    QIW: q∈W and
    ASSFMT[simp]: Q' = Q ∪ br-dsq δ q (Q, W)
                  W' = W − {q} ∪ (br-dsq δ q (Q, W) − Q)
    by (auto intro: br-step.cases[OF A(3)[simplified]])

  from A(2) have [simp]: finite Q
    by (auto simp add: br-invar-def)
  from A(2)[unfolded br-invar-def] have [simp]: finite W
    by (auto simp add: finite-subset)
  from A(1) have WNE: W≠{} by (unfold br-cond-def) auto

  note DSQSS = br-dsq-ss[OF A(2)[simplified] WNE QIW]
  {
    assume br-dsq δ q (Q,W) − Q = {}
    hence ?thesis using QIW
      by (simp add: br-termrel-def set-simps)
  } moreover {
    assume br-dsq δ q (Q,W) − Q ≠ {}
    hence Q ⊂ Q' by auto
    moreover from DSQSS A(2)[unfolded br-invar-def] have
      Q' ⊆ b-accessible δ
      by auto
    ultimately have ?thesis
      by (simp add: br-termrel-def)
  } ultimately show ?thesis by blast
qed

lemma br-invar-initial[simp]: finite δ ⟹ (br-initial δ)∈br-invar δ
  apply (auto simp add: br-initial-def br-invar-def br-iq-def)

  apply (case-tac r)
  apply (fastforce intro: b-accessible.intros)
  apply (fastforce elim!: bacc-step.cases)
  done
```

```
lemma br-invar-step:
  assumes [simp]: finite δ
  assumes A: Σ∈br-cond    Σ∈br-invar δ    (Σ,Σ')∈br-step δ
  shows Σ'∈br-invar δ
proof −
  obtain Q W Q' W' where SF[simp]: Σ=(Q, W)    Σ'=(Q', W')
    by (cases Σ, cases Σ', auto)
  obtain q where
    QIW: q∈W and
    ASSFMT[simp]: Q' = Q ∪ br-dsq δ q (Q, W)
                  W' = W − {q} ∪ (br-dsq δ q (Q, W) − Q)
    by (auto intro: br-step.cases[OF A(3)[simplified]])

  from A(1) have WNE: W≠{} by (unfold br-cond-def) auto

  have DSQSS: br-dsq δ q (Q,W) ⊆ b-accessible δ
    using br-dsq-ss[OF A(2)[simplified] WNE QIW] .

  show ?thesis
    apply (simp add: br-invar-def del: ASSFMT)
  proof (intro conjI)
    from A(2) have W ⊆ Q by (simp add: br-invar-def)
    thus W' ⊆ Q' by auto
  next
    from A(2) have Q ⊆ b-accessible δ by (simp add: br-invar-def)
    with DSQSS show Q' ⊆ b-accessible δ by auto
  next
    show bacc-step δ (Q' − W') ⊆ Q'
      apply (rule subsetI)
      apply (erule bacc-step.cases)
      apply (auto simp add: br-dsq-def)
      done
  next
    show finite Q' using A(2) by (simp add: br-invar-def br-dsq-def)
  qed
qed


lemma br-invar-final:
  ∀Σ. Σ∈wa-invar (br-algo δ) ∧ Σ∉wa-cond (br-algo δ)
   ⟶ fst Σ = b-accessible δ
  apply (simp add: br-invar-def br-cond-def br-algo-def)
  apply (auto intro: rev-subsetD[OF - b-accs-as-closed])
  done


theorem br-while-algo:
  assumes FIN[simp]: finite δ
```

shows *while-algo* (*br-algo* δ)
apply (*unfold-locales*)
apply (*simp-all add*: *br-algo-def br-invar-step br-invar-initial*
br-step-in-termrel)
apply (*rule-tac r*=*br-termrel* δ **in** *wf-subset*)
apply (*auto intro*: *br-step-in-termrel*)
done

lemma *bre-invar-final*:
$\forall \Sigma.\ \Sigma \in$*wa-invar* (*bre-algo Qi* δ) $\wedge$ $\Sigma \notin$*wa-cond* (*bre-algo Qi* δ)
$\longrightarrow$ ((*Qi*$\cap$*fst* Σ=$\{\}$) $\longleftrightarrow$ (*Qi* $\cap$ *b-accessible* δ = $\{\}$))
apply (*simp add*: *br-invar-def bre-cond-def bre-algo-def*)
apply *safe*
apply (*auto dest*!: *b-accs-as-closed*)
done

theorem *bre-while-algo*:
assumes *FIN*[*simp*]: *finite* δ
shows *while-algo* (*bre-algo Qi* δ)
apply (*unfold-locales*)
apply (*unfold bre-algo-def*)
apply (*auto simp add*: *br-invar-initial br-step-in-termrel*
intro: *br-invar-step*
dest: *rev-subsetD*[*OF - bre-cond-imp-br-cond*])
apply (*rule-tac r*=*br-termrel* δ **in** *wf-subset*)
apply (*auto intro*: *br-step-in-termrel*
dest: *rev-subsetD*[*OF - bre-cond-imp-br-cond*])
done

α'- Level Here, an optimization is added: For each rule, the algorithm now maintains a counter that counts the number of undiscovered states on the rules RHS. Whenever a new state is discovered, this counter is decremented for all rules where the state occurs on the RHS. The LHS states of rules where the counter falls to 0 are added to the worklist. The idea is that decrementing the counter is more efficient than checking whether all states on the rule's RHS have been discovered.

A similar algorithm is sketched in [2](Exercise 1.18).

type-synonym ('*Q*,'*L*) *br'-state* = '*Q set* $\times$ '*Q set* $\times$ (('*Q*,'*L*) *ta-rule* $\rightharpoonup$ *nat*)

— Abstraction to α-level
definition *br'*-α :: ('*Q*,'*L*) *br'-state* $\Rightarrow$ ('*Q*,'*L*) *br-state*
where *br'*-α = (λ(*Q*,*W*,*rcm*). (*Q*,*W*))

definition *br'-invar-add* :: ('*Q*,'*L*) *ta-rule set* $\Rightarrow$ ('*Q*,'*L*) *br'-state set*
where *br'-invar-add* δ == {(*Q*,*W*,*rcm*).
($\forall$ *r*$\in\delta$. *rcm r* = *Some* (*card* (*set* (*rhsq r*) $-$ (*Q* $-$ *W*)))) $\wedge$
{*lhs r* | *r*. *r*$\in\delta$ $\wedge$ *the* (*rcm r*) = *0*} $\subseteq$ *Q*

}

definition *br′-invar* :: (′*Q*,′*L*) *ta-rule set* ⇒ (′*Q*,′*L*) *br′-state set*
where *br′-invar* δ == *br′-invar-add* δ ∩ {Σ. *br′-α* Σ ∈ *br-invar* δ}

inductive-set *br′-step*
:: (′*Q*,′*L*) *ta-rule set* ⇒ ((′*Q*,′*L*) *br′-state* × (′*Q*,′*L*) *br′-state*) *set*
for δ **where**
⟦ *q*∈*W*;
Q′ = *Q* ∪ { *lhs r* | *r*. *r*∈δ ∧ *q* ∈ *set* (*rhsq r*) ∧ *the* (*rcm r*) ≤ *1* };
W′ = (*W*−{*q*})
∪ ({ *lhs r* | *r*. *r*∈δ ∧ *q* ∈ *set* (*rhsq r*) ∧ *the* (*rcm r*) ≤ *1* }
− *Q*);
!!*r*. *r*∈δ ⟹ *rcm′ r* = (*if q* ∈ *set* (*rhsq r*) *then*
Some (*the* (*rcm r*) − *1*)
else rcm r
)
⟧ ⟹ ((*Q*,*W*,*rcm*),(*Q′*,*W′*,*rcm′*)) ∈ *br′-step* δ

definition *br′-cond* :: (′*Q*,′*L*) *br′-state set*
where *br′-cond* == {(*Q*,*W*,*rcm*). *W*≠{}}
definition *bre′-cond* :: ′*Q set* ⇒ (′*Q*,′*L*) *br′-state set*
where *bre′-cond Qi* == {(*Q*,*W*,*rcm*). *W*≠{} ∧ (*Qi*∩*Q*={})}

inductive-set *br′-initial* :: (′*Q*,′*L*) *ta-rule set* ⇒ (′*Q*,′*L*) *br′-state set*
for δ **where**
⟦ !!*r*. *r*∈δ ⟹ *rcm r* = *Some* (*card* (*set* (*rhsq r*))) ⟧
⟹ (*br-iq* δ, *br-iq* δ, *rcm*)∈*br′-initial* δ

definition *br′-algo* δ == (|
wa-cond=*br′-cond*,
wa-step = *br′-step* δ,
wa-initial = *br′-initial* δ,
wa-invar = *br′-invar* δ
|)

definition *bre′-algo Qi* δ == (|
wa-cond=*bre′-cond Qi*,
wa-step = *br′-step* δ,
wa-initial = *br′-initial* δ,
wa-invar = *br′-invar* δ
|)

lemma *br′-step-invar*:
assumes *finite*[*simp*]: *finite* δ
assumes *INV*: Σ∈*br′-invar-add* δ *br′-α* Σ ∈ *br-invar* δ
assumes *STEP*: (Σ,Σ′) ∈ *br′-step* δ
shows Σ′∈*br′-invar-add* δ
proof −

obtain Q W *rcm* **where** [*simp*]: Σ=(Q,W,*rcm*)
by (*cases* Σ) *auto*
obtain Q' W' *rcm'* **where** [*simp*]: Σ'=(Q',W',*rcm'*)
by (*cases* Σ') *auto*

from *STEP* **obtain** q **where**
STEPF:
$q \in W$
$Q' = Q \cup \{$ *lhs* r | r. $r \in \delta \wedge q \in$ *set* (*rhsq* r) $\wedge$ *the* (*rcm* r) $\leq$ *1* $\}$
$W' = (W - \{q\})$
$\cup$ ($\{$ *lhs* r | r. $r \in \delta \wedge q \in$ *set* (*rhsq* r) $\wedge$ *the* (*rcm* r) $\leq$ *1* $\}$
$-$ Q)
!!r. $r \in \delta \Longrightarrow$ *rcm'* r = (*if* $q \in$ *set* (*rhsq* r) *then*
Some (*the* (*rcm* r) $-$ *1*)
else rcm r
)
by (*auto elim*: *br'-step.cases*)

from *INV*[*unfolded br'-invar-def br-invar-def br'-invar-add-def br'-α-def*,
simplified]
have *INV*:
($\forall r \in \delta$. *rcm* r = *Some* (*card* (*set* (*rhsq* r) $-$ (Q $-$ W))))
$\{$*lhs* r |r. $r \in \delta \wedge$ *the* (*rcm* r) = *0*$\}$ $\subseteq$ Q
$W \subseteq Q$
$Q \subseteq$ *b-accessible* δ
bacc-step δ (Q $-$ W) $\subseteq$ Q
finite Q
by *auto*

{
fix r
assume *A*: $r \in \delta$
with *INV*(*1*) **have** *RCMR*: *rcm* r = *Some* (*card* (*set* (*rhsq* r) $-$ (Q $-$ W)))
by *auto*

have *rcm'* r = *Some* (*card* (*set* (*rhsq* r) $-$ (Q' $-$ W')))
proof (*cases* $q \in$*set* (*rhsq* r))
case *False*
with *A STEPF*(*4*) **have** *rcm'* r = *rcm* r **by** *auto*
moreover from *STEPF INV*(*3*) *False* **have**
set (*rhsq* r) $-$ ($Q$$-$$W$) = *set* (*rhsq* r) $-$ ($Q'$$-$$W'$)
by *auto*
ultimately show *?thesis*
by (*simp add*: *RCMR*)
next
case *True*
with *A STEPF*(*4*) *RCMR* **have**
rcm' r = *Some* ((*card* (*set* (*rhsq* r) $-$ (Q $-$ W))) $-$ *1*)
by *simp*

```
    moreover from STEPF INV(3) True have
      set (rhsq r) − (Q−W) = insert q (set (rhsq r) − (Q'−W'))
      q∉(set (rhsq r) − (Q'−W'))
      by auto
    ultimately show ?thesis
      by (simp add: RCMR card-insert-disjoint')
  qed
 } moreover {
  fix r
  assume A: r∈δ    the (rcm' r) = 0
  have lhs r ∈ Q' proof (cases q∈set (rhsq r))
    case True
    with A(1) STEPF(4) have rcm' r = Some (the (rcm r) − 1) by auto
    with A(2) have the (rcm r) − 1 = 0 by auto
    hence the (rcm r) ≤ 1 by auto
    with STEPF(2) A(1) True show ?thesis by auto
  next
    case False
    with A(1) STEPF(4) have rcm' r = rcm r by auto
    with A(2) have the (rcm r) = 0 by auto
    with A(1) INV(2) have lhs r ∈ Q by auto
    with STEPF(2) show ?thesis by auto
  qed
 } ultimately show ?thesis
  by (auto simp add: br'-invar-add-def)
qed

lemma br'-invar-initial:
 br'-initial δ ⊆ br'-invar-add δ
 apply safe
 apply (erule br'-initial.cases)
 apply (unfold br'-invar-add-def)
 apply (auto simp add: br-iq-def)
 done

lemma br'-rcm-aux':
 ⟦ (Q,W,rcm)∈br'-invar δ; q∈W ⟧
  ⟹ {r ∈ δ. q ∈ set (rhsq r) ∧ the (rcm r) ≤ Suc 0}
      = {r∈δ. q∈set (rhsq r) ∧ set (rhsq r) ⊆ (Q − (W−{q}))}
proof (intro subsetI equalityI, goal-cases)
 case prems: (1 r)
 hence B: r∈δ    q∈set (rhsq r)    the (rcm r) ≤ Suc 0 by auto
 from B(1,3) prems(1)[unfolded br'-invar-def br'-invar-add-def] have
  CARD: card (set (rhsq r) − (Q − W)) ≤ Suc 0
  by auto
 from prems(1)[unfolded br'-invar-def br-invar-def br'-α-def] have WSQ: W⊆Q

  by auto
 have set (rhsq r) − (Q − W) = {q}
```

```
  proof −
    from B(2) prems(2) have R1: q∈set (rhsq r) − (Q − W) by auto
    moreover
    {
      fix x
      assume A: x≠q    x∈set (rhsq r) − (Q − W)
      with R1 have {x,q} ⊆ set (rhsq r) − (Q − W) by auto
      hence card {x,q} ≤ card (set (rhsq r) − (Q − W))
        by (auto simp add: card-mono)
      with CARD A(1) have False by auto
    }
    ultimately show ?thesis by auto
  qed
  with prems(2) WSQ have set (rhsq r) ⊆ Q − (W − {q}) by auto
  thus ?case using B(1,2) by auto
next
  case prems: (2 r)
  hence B: r∈δ    q∈set (rhsq r)    set (rhsq r) ⊆ Q − (W − {q}) by auto
  with prems(1)[unfolded br'-invar-def br'-invar-add-def
                         br'-α-def br-invar-def]
  have
    IC: W⊆Q    the (rcm r) = card (set (rhsq r) − (Q − W))
    by auto
  have set (rhsq r) − (Q − W) ⊆ {q} using B(2,3) IC(1) by auto
  from card-mono[OF - this] have the (rcm r) ≤ Suc 0 by (simp add: IC(2))
  with B(1,2) show ?case by auto
qed

lemma br'-rcm-aux:
  assumes A: (Q,W,rcm)∈br'-invar δ    q∈W
  shows {lhs r |r. r ∈ δ ∧ q ∈ set (rhsq r) ∧ the (rcm r) ≤ Suc 0}
        = {lhs r | r. r∈δ ∧ q∈set (rhsq r) ∧ set (rhsq r) ⊆ (Q − (W−{q}))}
proof −
  have {lhs r |r. r ∈ δ ∧ q ∈ set (rhsq r) ∧ the (rcm r) ≤ Suc 0}
       = lhs ‘ {r ∈ δ. q ∈ set (rhsq r) ∧ the (rcm r) ≤ Suc 0}
    by auto
  also from br'-rcm-aux'[OF A] have
    ... = lhs ‘ {r ∈ δ. q ∈ set (rhsq r) ∧ set (rhsq r) ⊆ Q − (W − {q})}
    by simp
  also have
    ... = {lhs r | r. r∈δ ∧ q∈set (rhsq r) ∧ set (rhsq r) ⊆ (Q − (W−{q}))}
    by auto
  finally show ?thesis .
qed

lemma br'-invar-QcD:
  (Q,W,rcm) ∈ br'-invar δ ⟹ {lhs r | r. r∈δ ∧ set (rhsq r) ⊆ (Q−W)} ⊆ Q
proof (safe)
  fix r
```

assume *A*: $(Q,W,rcm)\in br'\text{-}invar\ \delta \quad r\in\delta \quad set\ (rhsq\ r) \subseteq Q - W$
from *A(1)*[*unfolded br'-invar-def br'-invar-add-def br'-α-def br-invar-def*, *simplified*]
have
IC: $W \subseteq Q$
$finite\ Q$
$(\forall r\in\delta.\ rcm\ r = Some\ (card\ (set\ (rhsq\ r) - (Q - W))))$
$\{lhs\ r\ |r.\ r \in \delta \wedge the\ (rcm\ r) = 0\} \subseteq Q$ **by** *auto*
from *IC(3) A(2,3)* **have** $the\ (rcm\ r) = 0$ **by** *auto*
with *IC(4) A(2)* **show** $lhs\ r \in Q$ **by** *auto*
qed

lemma *br'-rcm-aux2*:
$[\![\ (Q,W,rcm)\in br'\text{-}invar\ \delta;\ q\in W\ ]\!]$
$\Longrightarrow Q \cup br\text{-}dsq\ \delta\ q\ (Q,W)$
$= Q \cup \{lhs\ r\ |r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$
apply (*simp only*: *br'-rcm-aux*)
apply (*unfold br-dsq-def*)
apply *simp*
apply (*frule br'-invar-QcD*)
apply *auto*
done

lemma *br'-rcm-aux3*:
$[\![\ (Q,W,rcm)\in br'\text{-}invar\ \delta;\ q\in W\ ]\!]$
$\Longrightarrow br\text{-}dsq\ \delta\ q\ (Q,W) - Q$
$= \{lhs\ r\ |r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\} - Q$
apply (*simp only*: *br'-rcm-aux*)
apply (*unfold br-dsq-def*)
apply *simp*
apply (*frule br'-invar-QcD*)
apply *auto*
done

lemma *br'-step-abs*:
$[\![$
$\Sigma\in br'\text{-}invar\ \delta;$
$(\Sigma,\Sigma') \in br'\text{-}step\ \delta$
$]\!] \Longrightarrow (br'\text{-}\alpha\ \Sigma,\ br'\text{-}\alpha\ \Sigma')\in br\text{-}step\ \delta$
apply (*cases* Σ, *cases* Σ', *simp*)
apply (*erule br'-step.cases*)
apply (*simp add*: *br'-α-def*)
apply (*rule-tac* $q=q$ **in** *br-step.intros*)
apply *simp*
apply (*simp only*: *br'-rcm-aux2*)
apply (*simp only*: *br'-rcm-aux3*)
done

lemma *br′-initial-abs*: *br′-α‘(br′-initial δ) = {br-initial δ}*
apply (*force simp add*: *br-initial-def br′-α-def*
elim: *br′-initial.cases*
intro: *br′-initial.intros*)
done

lemma *br′-cond-abs*: *Σ∈br′-cond ⟷ (br′-α Σ) ∈ br-cond*
by (*cases Σ*)
(*simp add*: *br′-cond-def br-cond-def br′-α-def image-Collect*
br′-algo-def br-algo-def)

lemma *bre′-cond-abs*: *Σ∈bre′-cond Qi ⟷ (br′-α Σ)∈bre-cond Qi*
by (*cases Σ*) (*simp add*: *bre′-cond-def bre-cond-def br′-α-def image-Collect*
bre′-algo-def bre-algo-def)

lemma *br′-invar-abs*: *br′-α‘br′-invar δ ⊆ br-invar δ*
by (*auto simp add*: *br′-invar-def*)

theorem *br′-pref-br*: *wa-precise-refine* (*br′-algo δ*) (*br-algo δ*) *br′-α*
apply *unfold-locales*
apply (*simp-all add*: *br′-algo-def br-algo-def*)
apply (*simp-all add*: *br′-cond-abs br′-step-abs br′-invar-abs br′-initial-abs*)
done

interpretation *br′-pref*: *wa-precise-refine br′-algo δ br-algo δ br′-α*
using *br′-pref-br* **.**

theorem *br′-while-algo*:
finite δ ⟹ while-algo (*br′-algo δ*)
apply (*rule br′-pref.wa-intro*)
apply (*simp add*: *br-while-algo*)
apply (*simp-all add*: *br′-algo-def br-algo-def*)
apply (*simp add*: *br′-invar-def*)
apply (*erule* (*3*) *br′-step-invar*)
apply (*simp add*: *br′-invar-initial*)
done

lemma *fst-br′-α*: *fst* (*br′-α s*) = *fst s* **by** (*cases s*) (*simp add*: *br′-α-def*)

lemmas *br′-invar-final* =
br′-pref.transfer-correctness[*OF br-invar-final*, *unfolded fst-br′-α*]

theorem *bre′-pref-br*: *wa-precise-refine* (*bre′-algo Qi δ*) (*bre-algo Qi δ*) *br′-α*
apply *unfold-locales*
apply (*simp-all add*: *bre′-algo-def bre-algo-def*)
apply (*simp-all add*: *bre′-cond-abs br′-step-abs br′-invar-abs br′-initial-abs*)
done

interpretation *bre′-pref*:

wa-precise-refine bre′-algo Qi δ bre-algo Qi δ br′-α
using *bre′-pref-br* **.**

theorem *bre′-while-algo*:
finite δ ⟹ while-algo (bre′-algo Qi δ)
apply (*rule bre′-pref.wa-intro*)
apply (*simp add*: *bre-while-algo*)
apply (*simp-all add*: *bre′-algo-def bre-algo-def*)
apply (*simp add*: *br′-invar-def*)
apply (*erule* (*3*) *br′-step-invar*)
apply (*simp add*: *br′-invar-initial*)
done

lemmas *bre′-invar-final* =
bre′-pref.transfer-correctness[*OF bre-invar-final*, *unfolded fst-br′-α*]

Implementing a Step In this paragraph, it is shown how to implement a step of the br'-algorithm by iteration over the rules that have the discovered state on their RHS.

definition *br′-inner-step*
:: (′*Q*,′*L*) *ta-rule* ⇒ (′*Q*,′*L*) *br′-state* ⇒ (′*Q*,′*L*) *br′-state*
where
br′-inner-step == λ*r* (*Q*,*W*,*rcm*). *let c*=*the* (*rcm r*) *in* (
if c≤*1 then insert* (*lhs r*) *Q else Q*,
if c≤*1* ∧ (*lhs r*) ∉ *Q then insert* (*lhs r*) *W else W*,
rcm (*r* ↦ (*c*−(*1*::*nat*)))
)

definition *br′-inner-invar*
:: (′*Q*,′*L*) *ta-rule set* ⇒ ′*Q* ⇒ (′*Q*,′*L*) *br′-state*
⇒ (′*Q*,′*L*) *ta-rule set* ⇒ (′*Q*,′*L*) *br′-state* ⇒ *bool*
where
br′-inner-invar rules q == λ(*Q*,*W*,*rcm*) *it* (*Q′*,*W′*,*rcm′*).
Q′ = *Q* ∪ { *lhs r* | *r*. *r*∈*rules*−*it* ∧ *the* (*rcm r*) ≤ *1* } ∧
W′ = (*W*−{*q*}) ∪ ({ *lhs r* | *r*. *r*∈*rules*−*it* ∧ *the* (*rcm r*) ≤ *1* } − *Q*) ∧
(∀ *r*. *rcm′ r* = (*if r*∈*rules*−*it then Some* (*the* (*rcm r*) − *1*) *else rcm r*))

lemma *br′-inner-invar-imp-final*:
⟦ *q*∈*W*; *br′-inner-invar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*) {} Σ′ ⟧
⟹ ((*Q*,*W*,*rcm*),Σ′) ∈ *br′-step δ*
apply (*unfold br′-inner-invar-def*)
apply *auto*
apply (*rule br′-step.intros*)
apply *assumption*
apply *auto*
done

lemma *br′-inner-invar-step*:
  ⟦ $q \in W$; *br′-inner-invar* $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ q $(Q, W - \{q\}, rcm)$ *it* Σ';
    $r \in it$; $it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\}$
  ⟧ ⟹ *br′-inner-invar* $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ q $(Q, W - \{q\}, rcm)$
      $(it - \{r\})$ (*br′-inner-step* r Σ')

  apply (*cases* Σ', *simp*)
  apply (*unfold br′-inner-invar-def br′-inner-step-def Let-def*)
  apply *auto*
  done

lemma *br′-inner-invar-initial*:
  ⟦ $q \in W$ ⟧ ⟹ *br′-inner-invar* $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ q $(Q, W - \{q\}, rcm)$
      $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ $(Q, W - \{q\}, rcm)$
  apply (*simp add*: *br′-inner-invar-def*)
  apply *auto*
  done

lemma *br′-inner-step-proof*:
  fixes $\alpha s :: '\Sigma \Rightarrow ('Q, 'L)$ *br′-state*
  fixes *cstep* :: $('Q, 'L)$ *ta-rule* ⇒ $'\Sigma \Rightarrow '\Sigma$
  fixes $\Sigma h :: '\Sigma$
  fixes *cinvar* :: $('Q, 'L)$ *ta-rule set* ⇒ $'\Sigma \Rightarrow$ *bool*

  assumes *iterable-set*: *set-iteratei* α *invar iteratei*
  assumes *invar-initial*: *cinvar* $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ Σh
  assumes *invar-step*:
    !!$it\ r\ \Sigma$. ⟦ $r \in it$; $it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\}$; *cinvar it* Σ ⟧
      ⟹ *cinvar* $(it - \{r\})$ (*cstep* r Σ)
  assumes *step-desc*:
    !!$it\ r\ \Sigma$. ⟦ $r \in it$; $it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\}$; *cinvar it* Σ ⟧
      ⟹ $\alpha s\ (cstep\ r\ \Sigma) = br'\text{-}inner\text{-}step\ r\ (\alpha s\ \Sigma)$
  assumes *it-set-desc*: *invar it-set* α *it-set* $= \{r \in \delta.\ q \in set\ (rhsq\ r)\}$

  assumes *QIW*[*simp*]: $q \in W$

  assumes Σ-*desc*[*simp*]: $\alpha s\ \Sigma = (Q, W, rcm)$
  assumes Σh-*desc*[*simp*]: $\alpha s\ \Sigma h = (Q, W - \{q\}, rcm)$

  shows $(\alpha s\ \Sigma, \alpha s\ (iteratei\ it\text{-}set\ (\lambda\text{-}.\ True)\ cstep\ \Sigma h)) \in br'\text{-}step\ \delta$
proof −
  interpret *set-iteratei* α *invar iteratei* **by** *fact*

  show *?thesis*
    apply (*rule-tac*
      $I = \lambda it\ \Sigma.$ *cinvar it* Σ
          ∧ *br′-inner-invar* $\{r \in \delta.\ q \in set\ (rhsq\ r)\}$ q $(Q, W - \{q\}, rcm)$
              it $(\alpha s\ \Sigma)$

```
    in iterate-rule-P)
  apply (simp-all
    add: it-set-desc invar-initial br'-inner-invar-initial invar-step
      step-desc br'-inner-invar-step)
  apply (rule br'-inner-invar-imp-final)
  apply (rule QIW)
  apply simp
  done
qed
```

Computing Witnesses The algorithm is now refined further, such that it stores, for each discovered state, a witness for non-emptiness, i.e. a tree that is accepted with the discovered state.

— A map from states to trees has the witness-property, if it maps states to trees that are accepted with that state:

```
definition witness-prop δ m == ∀ q t. m q = Some t ⟶ accs δ t q
```

— Construct a witness for the LHS of a rule, provided that the map contains witnesses for all states on the RHS:

```
definition construct-witness
  :: ('Q ⇀ 'L tree) ⇒ ('Q,'L) ta-rule ⇒ 'L tree
  where
  construct-witness Q r == NODE (rhsl r) (List.map (λq. the (Q q)) (rhsq r))

lemma witness-propD: ⟦witness-prop δ m; m q = Some t⟧ ⟹ accs δ t q
  by (auto simp add: witness-prop-def)

lemma construct-witness-correct:
  ⟦ witness-prop δ Q; r∈δ; set (rhsq r) ⊆ dom Q ⟧
    ⟹ accs δ (construct-witness Q r) (lhs r)
  apply (unfold construct-witness-def witness-prop-def)
  apply (cases r)
  apply simp
  apply (erule accs.intros)
  apply (auto dest: nth-mem)
  done

lemma construct-witness-eq:
  ⟦ Q |` set (rhsq r) = Q' |` set (rhsq r) ⟧ ⟹
    construct-witness Q r = construct-witness Q' r
  apply (unfold construct-witness-def)
  apply auto
  apply (subgoal-tac Q x = Q' x)
  apply (simp)
  apply (drule-tac x=x in fun-cong)
  apply auto
  done
```

The set of discovered states is refined by a map from discovered states to their witnesses:

type-synonym $('Q,'L)$ *brw-state* $= ('Q \rightharpoonup 'L\ tree) \times 'Q\ set \times (('Q,'L)\ ta\text{-}rule \rightharpoonup nat)$

definition *brw-α* :: $('Q,'L)$ *brw-state* $\Rightarrow$ $('Q,'L)$ *br′-state*
where *brw-α* $= (\lambda(Q,W,rcm).\ (dom\ Q, W, rcm))$

definition *brw-invar-add* :: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $('Q,'L)$ *brw-state set*
where *brw-invar-add* $\delta == \{(Q,W,rcm).\ witness\text{-}prop\ \delta\ Q\}$

definition *brw-invar* $\delta ==$ *brw-invar-add* $\delta \cap \{s.\ brw\text{-}\alpha\ s \in br'\text{-}invar\ \delta\}$

inductive-set *brw-step*
:: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $(('Q,'L)\ brw\text{-}state \times ('Q,'L)\ brw\text{-}state)\ set$
for δ **where**
$[\![$
$q \in W$;
$dsqr = \{\ r \in \delta.\ q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq 1\ \}$;
$dom\ Q' = dom\ Q \cup lhs`dsqr$;
$!!q\ t.\ Q'\ q = Some\ t \Longrightarrow Q\ q = Some\ t$
$\vee\ (\exists\, r \in dsqr.\ q = lhs\ r \wedge t = construct\text{-}witness\ Q\ r)$;
$W' = (W - \{q\}) \cup (lhs`dsqr - dom\ Q)$;
$!!r.\ r \in \delta \Longrightarrow rcm'\ r = (\ if\ q \in set\ (rhsq\ r)\ then$
$Some\ (the\ (rcm\ r) - 1)$
$else\ rcm\ r$
$)$
$]\!] \Longrightarrow ((Q,W,rcm),(Q',W',rcm')) \in brw\text{-}step\ \delta$

definition *brw-cond* :: $'Q\ set \Rightarrow ('Q,'L)\ brw\text{-}state\ set$
where *brw-cond Qi* $== \{(Q,W,rcm).\ W \neq \{\} \wedge (Qi \cap dom\ Q = \{\})\}$

inductive-set *brw-iq* :: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $('Q \rightharpoonup 'L\ tree)\ set$
for δ **where**
$[\![$
$\forall\, q\ t.\ Q\ q = Some\ t \longrightarrow (\exists\, r \in \delta.\ rhsq\ r = [] \wedge q = lhs\ r$
$\wedge\ t = NODE\ (rhsl\ r)\ [])$;
$\forall\, r \in \delta.\ rhsq\ r = [] \longrightarrow Q\ (lhs\ r) \neq None$
$]\!] \Longrightarrow Q \in brw\text{-}iq\ \delta$

inductive-set *brw-initial* :: $('Q,'L)$ *ta-rule set* $\Rightarrow$ $('Q,'L)$ *brw-state set*
for δ **where**
$[\![\ !!r.\ r \in \delta \Longrightarrow rcm\ r = Some\ (card\ (set\ (rhsq\ r)))$; $Q \in brw\text{-}iq\ \delta\ ]\!]$
$\Longrightarrow (Q,\ br\text{-}iq\ \delta,\ rcm) \in brw\text{-}initial\ \delta$

definition *brw-algo Qi* $\delta ==$ $(\!|$
wa-cond=brw-cond Qi,

```
  wa-step = brw-step δ,
  wa-initial = brw-initial δ,
  wa-invar = brw-invar δ
⦈

lemma brw-cond-abs: Σ∈brw-cond Qi ⟷ (brw-α Σ)∈bre′-cond Qi
  apply (cases Σ)
  apply (simp add: brw-cond-def bre′-cond-def brw-α-def)
  done

lemma brw-initial-abs: Σ∈brw-initial δ ⟹ brw-α Σ ∈ br′-initial δ
  apply (cases Σ, simp)
  apply (erule brw-initial.cases)
  apply (erule brw-iq.cases)
  apply (auto simp add: brw-α-def)
  apply (subgoal-tac dom Qa = br-iq δ)
  apply simp
  apply (rule br′-initial.intros)
  apply auto [1]
  apply (force simp add: br-iq-def)
  done


lemma brw-invar-initial: brw-initial δ ⊆ brw-invar-add δ
  apply safe
  apply (unfold brw-invar-add-def)
  apply (auto simp add: witness-prop-def)
  apply (erule brw-initial.cases)
  apply (erule brw-iq.cases)
  apply auto
proof goal-cases
  case prems: (1 q t rcm Q)
  from prems(3)[rule-format, OF prems(1)] obtain r where
    [simp]: r∈δ    rhsq r = []    q=lhs r    t=NODE (rhsl r) []
    by blast
  have RF[simplified]: r=((lhs r) → (rhsl r) (rhsq r)) by (cases r) simp
  show ?case
    apply (simp)
    apply (rule accs.intros)
    apply (subst RF[symmetric])
    apply auto
    done
qed

lemma brw-step-abs:
  ⟦ (Σ,Σ′)∈brw-step δ ⟧ ⟹ (brw-α Σ, brw-α Σ′)∈br′-step δ
  apply (cases Σ, cases Σ′, simp)
  apply (erule brw-step.cases)
  apply (simp add: brw-α-def)
```

```
  apply hypsubst
  apply (rule br'-step.intros)
  apply assumption
  apply auto
  done

lemma brw-step-invar:
  assumes FIN[simp]: finite δ
  assumes INV: Σ∈brw-invar-add δ and BR'INV: brw-α Σ ∈ br'-invar δ
  assumes STEP: (Σ,Σ') ∈ brw-step δ
  shows Σ'∈brw-invar-add δ
proof −
  obtain Q W rcm Q' W' rcm' where
    [simp]: Σ=(Q,W,rcm)    Σ'=(Q',W',rcm')
    by (cases Σ, cases Σ') force

  from INV have WP: witness-prop δ Q
    by (simp-all add: brw-invar-add-def)

  obtain qw dsqr where SPROPS:
    dsqr = {r ∈ δ. qw ∈ set (rhsq r) ∧ the (rcm r) ≤ 1}
    qw∈W
    dom Q' = dom Q ∪ lhs ‘ dsqr
    !!q t. Q' q = Some t ⟹ Q q = Some t
                         ∨ (∃ r∈dsqr. q=lhs r ∧ t=construct-witness Q r)
    by (auto intro: brw-step.cases[OF STEP[simplified]])
  from br'-rcm-aux'[OF BR'INV[unfolded brw-α-def, simplified] SPROPS(2)]
have
    DSQR-ALT: dsqr = {r ∈ δ. qw ∈ set (rhsq r)
                         ∧ set (rhsq r) ⊆ dom Q − (W − {qw})}
    by (simp add: SPROPS(1))
  have witness-prop δ Q'
  proof (unfold witness-prop-def, safe)
    fix q t
    assume A: Q' q = Some t

    from SPROPS(4)[OF A] have
      Q q = Some t ∨ (∃ r∈dsqr. q = lhs r ∧ t = construct-witness Q r) .
    moreover {
      assume C: Q q = Some t
      from witness-propD[OF WP, OF C] have accs δ t q .
    } moreover {
      fix r
      assume r∈dsqr and [simp]: q=lhs r    t=construct-witness Q r
      from ‹r∈dsqr› have 1: r∈δ    set (rhsq r) ⊆ dom Q
        by (auto simp add: DSQR-ALT)
      from construct-witness-correct[OF WP 1] have accs δ t q by simp
    } ultimately show accs δ t q by blast
  qed
```

thus *?thesis* **by** (*simp add*: *brw-invar-add-def*)
qed

theorem *brw-pref-bre'*: *wa-precise-refine* (*brw-algo Qi δ*) (*bre'-algo Qi δ*) *brw-α*
apply (*unfold-locales*)
apply (*simp-all add*: *brw-algo-def bre'-algo-def*)
apply (*auto simp add*: *brw-cond-abs brw-step-abs brw-initial-abs brw-invar-def*)
done

interpretation *brw-pref*:
wa-precise-refine brw-algo Qi δ bre'-algo Qi δ brw-α
using *brw-pref-bre'* **.**

theorem *brw-while-algo*: *finite δ* ⟹ *while-algo* (*brw-algo Qi δ*)
apply (*rule brw-pref.wa-intro*)
apply (*simp add*: *bre'-while-algo*)
apply (*simp-all add*: *brw-algo-def bre'-algo-def*)
apply (*simp add*: *brw-invar-def*)
apply (*auto intro*: *brw-step-invar simp add*: *brw-invar-initial*)
done

lemma *fst-brw-α*: *fst* (*brw-α s*) = *dom* (*fst s*)
by (*cases s*) (*simp add*: *brw-α-def*)

theorem *brw-invar-final*:
∀ *sc*. *sc* ∈ *wa-invar* (*brw-algo Qi δ*) ∧ *sc* ∉ *wa-cond* (*brw-algo Qi δ*)
⟶ (*Qi* ∩ *dom* (*fst sc*) = {}) = (*Qi* ∩ *b-accessible δ* = {})
∧ (*witness-prop δ* (*fst sc*))
apply (*intro conjI allI impI*)
using *brw-pref.transfer-correctness*[*OF bre'-invar-final*, *unfolded fst-brw-α*]
apply *blast*
apply (*auto simp add*: *brw-algo-def brw-invar-def brw-invar-add-def*)
done

Implementing a Step **inductive-set** *brw-inner-step*
:: ('*Q*,'*L*) *ta-rule* ⇒ (('*Q*,'*L*) *brw-state* × ('*Q*,'*L*) *brw-state*) *set*
for *r* **where**
⟦ *c* = *the* (*rcm r*); Σ = (*Q*,*W*,*rcm*); Σ'=(*Q'*,*W'*,*rcm'*);
if c≤*1* ∧ (*lhs r*) ∉ *dom Q then*
Q' = *Q*(*lhs r* ↦ *construct-witness Q r*)
else Q' = *Q*;
if c≤*1* ∧ (*lhs r*) ∉ *dom Q then*
W' = *insert* (*lhs r*) *W*
else W' = *W*;
rcm' = *rcm* (*r* ↦ (*c*−(*1*::*nat*)))
⟧ ⟹ (Σ,Σ')∈*brw-inner-step r*

definition *brw-inner-invar*
:: ('*Q*,'*L*) *ta-rule set* ⇒ '*Q* ⇒ ('*Q*,'*L*) *brw-state* ⇒ ('*Q*,'*L*) *ta-rule set*

$\Rightarrow$ ('Q,'L) *brw-state* $\Rightarrow$ *bool*
where
brw-inner-invar rules q == λ(Q,W,*rcm*) *it* (Q',W',*rcm'*).
(*br'-inner-invar rules q* (*brw-*α (Q,W,*rcm*)) *it* (*brw-*α (Q',W',*rcm'*)) $\wedge$
($Q'|`dom\ Q = Q$) $\wedge$
(*let dsqr* = { $r \in rules - it$. *the* (*rcm r*) $\leq$ *1* } *in*
($\forall q\ t$. $Q'\ q$ = *Some t* $\longrightarrow$ ($Q\ q$ = *Some t*
$\vee$ ($Q\ q$ = *None* $\wedge$ ($\exists r \in dsqr$. q=*lhs r* $\wedge$ t=*construct-witness Q r*))
)
)))

lemma *brw-inner-step-abs*:
(Σ,Σ')$\in$*brw-inner-step r* $\Longrightarrow$ *br'-inner-step r* (*brw-*α Σ) = *brw-*α Σ'
apply (*erule brw-inner-step.cases*)
apply (*unfold br'-inner-step-def brw-*α*-def Let-def*)
apply *auto*
done

lemma *brw-inner-invar-imp-final*:
⟦ $q \in W$; *brw-inner-invar* {$r \in \delta$. $q \in$*set* (*rhsq r*)} q (Q,$W-\{q\}$,*rcm*) {} Σ' ⟧
$\Longrightarrow$ ((Q,W,*rcm*),Σ') $\in$ *brw-step* δ
apply (*unfold brw-inner-invar-def br'-inner-invar-def brw-*α*-def*)
apply (*auto simp add*: *Let-def*)
apply (*rule brw-step.intros*)
apply *assumption*
apply (*rule refl*)
apply *auto*
done

lemma *brw-inner-invar-step*:
assumes *INVI*: (Q,W,*rcm*)$\in$*brw-invar* δ
assumes *A*: $q \in W$ $r \in it$ $it \subseteq \{r \in \delta.\ q \in$*set* (*rhsq r*)$\}$
assumes *INVH*: *brw-inner-invar* {$r \in \delta$. $q \in$*set* (*rhsq r*)} q (Q,$W-\{q\}$,*rcm*) *it* Σh
assumes *STEP*: (Σh,Σ')$\in$*brw-inner-step r*
shows *brw-inner-invar* {$r \in \delta$. $q \in$*set* (*rhsq r*)} q (Q,$W-\{q\}$,*rcm*) (*it*$-\{r\}$) Σ'
proof −
from *INVI* **have** *BR'-INV*: (*dom Q*,W,*rcm*)$\in$*br'-invar* δ
by (*simp add*: *brw-invar-def brw-*α*-def*)

obtain *c Qh Wh rcmh Q' W' rcm'* **where**
SIGMAF[*simp*]: Σh=(Qh,Wh,*rcmh*) Σ'=(Q',W',*rcm'*) **and**
CF[*simp*]: *c* = *the* (*rcmh r*) **and**
SF: *if* $c \leq 1 \wedge$ (*lhs r*) $\notin$ *dom Qh then*
Q' = Qh(*lhs r* $\mapsto$ (*construct-witness Qh r*))
else Q' = Qh

```
      if c≤1 ∧ (lhs r) ∉ dom Qh then
         W' = insert (lhs r) Wh
       else W' = Wh

      rcm' = rcmh ( r ↦ (c−(1::nat)))
  by (blast intro: brw-inner-step.cases[OF STEP])
let ?rules = {r∈δ. q∈set (rhsq r)}
let ?dsqr = λit. { r∈?rules − it. the (rcm r) ≤ 1 }
from INVH have INVHF:
  br'-inner-invar ?rules q (dom Q, W−{q}, rcm) (it) (dom Qh,Wh,rcmh)
  Qh|`dom Q = Q
  (∀ q t. Qh q = Some t ⟶ (Q q = Some t
      ∨ (Q q = None ∧ (∃ r∈?dsqr it. q=lhs r ∧ t=construct-witness Q r))
                          )
  )
  by (auto simp add: brw-inner-invar-def Let-def brw-α-def)
from INVHF(1)[unfolded br'-inner-invar-def] have INV'HF:
  dom Qh = dom Q ∪ lhs`?dsqr it
  (∀ r. rcmh r = (if r ∈ ?rules − it then
                    Some (the (rcm r) − 1)
                  else rcm r))
  by auto
from brw-inner-step-abs[OF STEP]
     br'-inner-invar-step[OF A(1) INVHF(1) A(2,3)] have
 G1: br'-inner-invar ?rules q (dom Q, W−{q}, rcm) (it−{r}) (dom Q',W',rcm')
  by (simp add: brw-α-def)
moreover have
  (∀ q t. Q' q = Some t ⟶ (Q q = Some t
     ∨ ( Q q = None
         ∧ (∃ r∈?dsqr (it−{r}). q=lhs r ∧ t=construct-witness Q r)
       )
                          )
  ) (is ?G1)

  Q'|`dom Q = Q (is ?G2)
proof −
  {
    assume C: ¬ c≤1 ∨ lhs r ∈ dom Qh
    with SF have Q'=Qh by auto
    with INVHF(2,3) have ?G1 ?G2 by auto
  } moreover {
    assume C: c≤1    lhs r∉ dom Qh
    with SF have Q'F: Q'=Qh(lhs r ↦ (construct-witness Qh r)) by auto
   from C(2) INVHF(2) INV'HF(1) have G2: ?G2 by (auto simp add: Q'F)
    from C(1) INV'HF A have
      RI: r∈?dsqr (it−{r}) and
      DSS: dom Q ⊆ dom Qh
      by (auto)
```

```
      from br'-rcm-aux'[OF BR'-INV A(1)] RI have
        RDQ: set (rhsq r) ⊆ dom Q
        by auto
      with INVHF(2) have Qh |` set (rhsq r) = Q |` set (rhsq r)
        by (blast intro: restrict-map-subset-eq)
      hence [simp]: construct-witness Qh r = construct-witness Q r
        by (blast dest: construct-witness-eq)

       from DSS C(2) have [simp]: Q (lhs r) = None     Qh (lhs r) = None by
auto
      have G1: ?G1
      proof (intro allI impI, goal-cases)
        case prems: (1 q t)
        {
          assume [simp]: q=lhs r
          from prems Q'F have [simp]: t = (construct-witness Qh r) by simp
          from RI have ?case by auto
        } moreover {
          assume q≠lhs r
          with Q'F prems have Qh q = Some t by auto
          with INVHF(3) have ?case by auto
        } ultimately show ?case by blast
      qed
      note G1 G2
    } ultimately show ?G1 ?G2 by blast+
  qed
  ultimately show ?thesis
    by (unfold brw-inner-invar-def Let-def brw-α-def) auto
qed


lemma brw-inner-invar-initial:
  ⟦q∈W⟧ ⟹ brw-inner-invar {r∈δ. q∈set (rhsq r)} q (Q,W−{q},rcm)
                           {r∈δ. q∈set (rhsq r)} (Q,W−{q},rcm)
  by (simp add: brw-inner-invar-def br'-inner-invar-initial brw-α-def)

theorem brw-inner-step-proof:
  fixes αs :: 'Σ ⇒ ('Q,'L) brw-state
  fixes cstep :: ('Q,'L) ta-rule ⇒ 'Σ ⇒ 'Σ
  fixes Σh :: 'Σ
  fixes cinvar :: ('Q,'L) ta-rule set ⇒ 'Σ ⇒ bool

  assumes set-iterate: set-iteratei α invar iteratei
  assumes invar-start: (αs Σ)∈brw-invar δ
  assumes invar-initial: cinvar {r∈δ. q∈set (rhsq r)} Σh
  assumes invar-step:
    !!it r Σ. ⟦ r∈it; it ⊆ {r∈δ. q∈set (rhsq r)}; cinvar it Σ ⟧
              ⟹ cinvar (it−{r}) (cstep r Σ)
  assumes step-desc:
```

```
  !!it r Σ. ⟦ r∈it; it⊆{r∈δ. q∈set (rhsq r)}; cinvar it Σ ⟧
            ⟹ (αs Σ, αs (cstep r Σ)) ∈ brw-inner-step r
  assumes it-set-desc: invar it-set    α it-set = {r∈δ. q∈set (rhsq r)}

  assumes QIW[simp]: q∈W

  assumes Σ-desc[simp]: αs Σ = (Q,W,rcm)
  assumes Σh-desc[simp]: αs Σh = (Q,W−{q},rcm)

  shows (αs Σ, αs (iteratei it-set (λ-. True) cstep Σh))∈brw-step δ
proof −
  interpret set-iteratei α invar iteratei by fact

  show ?thesis
    apply (rule-tac
      I=λit Σ. cinvar it Σ ∧ brw-inner-invar {r∈δ. q∈set (rhsq r)} q
                                            (Q,W−{q},rcm) it (αs Σ)
      in iterate-rule-P)
    apply (auto
      simp add: it-set-desc invar-initial brw-inner-invar-initial invar-step
            step-desc brw-inner-invar-step[OF invar-start[simplified]]
            brw-inner-invar-imp-final[OF QIW])
    done
qed
```

4.3 Product Automaton

The forward-reduced product automaton can be described as a state-space exploration problem.

In this section, the DFS-algorithm for state-space exploration (cf. Theory *Collections-Examples.Exploration* in the Isabelle Collections Framework) is refined to compute the product automaton.

```
type-synonym ('Q1,'Q2,'L) frp-state =
  ('Q1×'Q2) set × ('Q1×'Q2) list × (('Q1×'Q2),'L) ta-rule set

definition frp-α :: ('Q1,'Q2,'L) frp-state ⇒ ('Q1×'Q2) dfs-state
  where frp-α S == let (Q,W,δ)=S in (Q, W)

definition frp-invar-add δ1 δ2 ==
  { (Q,W,δd). δd = { r. r∈δ-prod δ1 δ2 ∧ lhs r ∈ Q − set W} }

definition frp-invar
  :: ('Q1, 'L) tree-automaton-rec ⇒ ('Q2, 'L) tree-automaton-rec
     ⇒ ('Q1,'Q2,'L) frp-state set
  where frp-invar T1 T2 ==
  frp-invar-add (ta-rules T1) (ta-rules T2)
  ∩ { s. frp-α s ∈ dfs-invar (ta-initial T1 × ta-initial T2)
                             (f-succ (δ-prod (ta-rules T1) (ta-rules T2))) }
```

inductive-set *frp-step*
  :: (*'Q1*,*'L*) *ta-rule set* ⇒ (*'Q2*,*'L*) *ta-rule set*
     ⇒ ((*'Q1*,*'Q2*,*'L*) *frp-state* × (*'Q1*,*'Q2*,*'L*) *frp-state*) *set*
  for *δ1 δ2* **where**
  ⟦ *W*=(*q1*,*q2*)#*Wtl*;
     distinct Wn;
     set Wn = *f-succ* (*δ-prod δ1 δ2*) `` {(*q1*,*q2*)} − *Q*;
     W'=*Wn*@*Wtl*;
     Q'=*Q* ∪ *f-succ* (*δ-prod δ1 δ2*) `` {(*q1*,*q2*)};
     δd'=*δd* ∪ {*r*∈*δ-prod δ1 δ2*. *lhs r* = (*q1*,*q2*) }
  ⟧ ⟹ ((*Q*,*W*,*δd*),(*Q'*,*W'*,*δd'*))∈*frp-step δ1 δ2*

inductive-set *frp-initial* :: *'Q1 set* ⇒ *'Q2 set* ⇒ (*'Q1*,*'Q2*,*'L*) *frp-state set*
  for *Q10 Q20* **where**
  ⟦ *distinct W*; *set W* = *Q10*×*Q20* ⟧ ⟹ (*Q10*×*Q20*,*W*,{}) ∈ *frp-initial Q10 Q20*

definition *frp-cond* :: (*'Q1*,*'Q2*,*'L*) *frp-state set* **where**
  frp-cond == {(*Q*,*W*,*δd*). *W*≠[]}

definition *frp-algo T1 T2* == (|
  wa-cond = *frp-cond*,
  wa-step = *frp-step* (*ta-rules T1*) (*ta-rules T2*),
  wa-initial = *frp-initial* (*ta-initial T1*) (*ta-initial T2*),
  wa-invar = *frp-invar T1 T2*
|)

  — The algorithm refines the DFS-algorithm
theorem *frp-pref-dfs*:
  wa-precise-refine (*frp-algo T1 T2*)
    (*dfs-algo* (*ta-initial T1* × *ta-initial T2*)
            (*f-succ* (*δ-prod* (*ta-rules T1*) (*ta-rules T2*))))
    frp-α
  apply *unfold-locales*
  apply (*auto simp add*: *frp-algo-def frp-α-def frp-cond-def dfs-algo-def*
                    dfs-cond-def frp-invar-def
    elim!: *frp-step.cases frp-initial.cases*
    intro: *dfs-step.intros dfs-initial.intros*
)
  done

interpretation *frp-ref*: *wa-precise-refine* (*frp-algo T1 T2*)
              (*dfs-algo* (*ta-initial T1* × *ta-initial T2*)
                      (*f-succ* (*δ-prod* (*ta-rules T1*) (*ta-rules T2*))))
              frp-α **using** *frp-pref-dfs* **.**

— The algorithm is a well-defined while-algorithm
theorem *frp-while-algo*:
  assumes *TA*: *tree-automaton T1*

```
          tree-automaton T2
  shows while-algo (frp-algo T1 T2)
proof −
  interpret t1: tree-automaton T1 by fact
  interpret t2: tree-automaton T2 by fact

  have finite: finite ((f-succ (δ-prod (ta-rules T1) (ta-rules T2)))*
            `` (ta-initial T1 × ta-initial T2))
  proof −
    have ((f-succ (δ-prod (ta-rules T1) (ta-rules T2)))*
              `` (ta-initial T1 × ta-initial T2))
         ⊆ ((ta-initial T1 × ta-initial T2)
           ∪ δ-states (δ-prod (ta-rules T1) (ta-rules T2)))
      apply rule
      apply (drule f-accessible-subset[unfolded f-accessible-def])
      apply auto
      done
    moreover have finite ...
      by auto
    ultimately show ?thesis by (simp add: finite-subset)
  qed

  show ?thesis
    apply (rule frp-ref.wa-intro)
    apply (rule dfs-while-algo[OF finite])
    apply (simp add: frp-algo-def dfs-algo-def frp-invar-def)
    apply (auto simp add: dfs-algo-def frp-algo-def frp-α-def
                          dfs-α-def frp-invar-add-def dfs-invar-def
                          dfs-invar-add-def sse-invar-def
                elim!: frp-step.cases) [1]
    apply (force simp add: frp-algo-def frp-invar-add-def
                 elim!: frp-initial.cases)
    done
qed
```

— If the algorithm terminates, the forward reduced product automaton can be constructed from the result

```
theorem frp-inv-final:
  ∀ s. s∈wa-invar (frp-algo T1 T2) ∧ s∉wa-cond (frp-algo T1 T2)
       ⟶ (case s of (Q,W,δ d) ⇒
            (| ta-initial = ta-initial T1 × ta-initial T2,
               ta-rules = δ d
            |) = ta-fwd-reduce (ta-prod T1 T2))
  apply (intro allI impI)
  apply (case-tac s)
  apply simp
  apply (simp add: ta-reduce-def ta-prod-def frp-algo-def)
```

proof −
fix *Q W δd*
assume *A*: (*Q*,*W*,*δd*)∈*frp-invar T1 T2* ∧ (*Q*,*W*,*δd*)∉*frp-cond*

from *frp-ref.transfer-correctness*[*OF dfs-invar-final*,
unfolded frp-algo-def, *simplified*,
rule-format, *OF A*]
have [*simp*]: *Q* = *f-accessible* (*δ-prod* (*ta-rules T1*) (*ta-rules T2*))
(*ta-initial T1* × *ta-initial T2*)
by (*simp add*: *f-accessible-def dfs-α-def frp-α-def*)

from *A* **show** *δd* = *reduce-rules*
(*δ-prod* (*ta-rules T1*) (*ta-rules T2*))
(*f-accessible* (*δ-prod* (*ta-rules T1*) (*ta-rules T2*))
(*ta-initial T1* × *ta-initial T2*))
apply (*auto simp add*: *reduce-rules-def f-accessible-def frp-invar-def*
frp-invar-add-def frp-α-def frp-cond-def)
apply (*case-tac x*)
apply (*auto dest*: *rtrancl-into-rtrancl intro*: *f-succ.intros*)
done
qed

end

5 Executable Implementation of Tree Automata

theory *Ta-impl*
imports
Main
Collections.CollectionsV1
Ta AbsAlgo
HOL−Library.Code-Target-Numeral
begin

In this theory, an effcient executable implementation of non-deterministic tree automata and basic algorithms is defined.

The algorithms use red-black trees to represent sets of states or rules where appropriate.

5.1 Prelude

— Make rules hashable
instantiation *ta-rule* :: (*hashable*,*hashable*) *hashable*
begin
fun *hashcode-of-ta-rule*
:: (*'Q1*::*hashable*,*'Q2*::*hashable*) *ta-rule* ⇒ *hashcode*
where

hashcode-of-ta-rule (*q* → *f qs*) = *hashcode q* + *hashcode f* + *hashcode qs*

definition [*simp*]: *hashcode* = *hashcode-of-ta-rule*

definition *def-hashmap-size*::((*'a*,*'b*) *ta-rule itself* ⇒ *nat*) == (λ-. *32*)

instance
by (*intro-classes*)(*auto simp add*: *def-hashmap-size-ta-rule-def*)
end

— Make wrapped states hashable
instantiation *ustate-wrapper* :: (*hashable*,*hashable*) *hashable*
begin
definition *hashcode x* == (*case x of USW1 a* ⇒ *2* ∗ *hashcode a* | *USW2 b* ⇒ *2* ∗ *hashcode b* + *1*)
definition *def-hashmap-size* = (λ- :: ((*'a*,*'b*) *ustate-wrapper*) *itself*. *def-hashmap-size TYPE*(*'a*) + *def-hashmap-size TYPE*(*'b*))

instance using *def-hashmap-size*[**where** *?'a*=*'a*] *def-hashmap-size*[**where** *?'a*=*'b*]
by(*intro-classes*)(*simp-all add*: *bounded-hashcode-bounds def-hashmap-size-ustate-wrapper-def split*: *ustate-wrapper.split*)

end

5.1.1 Ad-Hoc instantiations of generic Algorithms

setup *Locale-Code.open-block*
interpretation *hll-idx*: *build-index-loc hm-ops ls-ops ls-ops* **by** *unfold-locales*
interpretation *ll-set-xy*: *g-set-xy-loc ls-ops ls-ops*
by *unfold-locales*

interpretation *lh-set-xx*: *g-set-xx-loc ls-ops hs-ops*
by *unfold-locales*
interpretation *lll-iflt-cp*: *inj-image-filter-cp-loc ls-ops ls-ops ls-ops*
by *unfold-locales*
interpretation *hhh-cart*: *cart-loc hs-ops hs-ops hs-ops* **by** *unfold-locales*
interpretation *hh-set-xy*: *g-set-xy-loc hs-ops hs-ops*
by *unfold-locales*

interpretation *llh-set-xyy*: *g-set-xyy-loc ls-ops ls-ops hs-ops*
by *unfold-locales*

interpretation *hh-map-to-nat*: *map-to-nat-loc hs-ops hm-ops* **by** *unfold-locales*
interpretation *hh-set-xy*: *g-set-xy-loc hs-ops hs-ops* **by** *unfold-locales*
interpretation *lh-set-xy*: *g-set-xy-loc ls-ops hs-ops* **by** *unfold-locales*
interpretation *hh-set-xx*: *g-set-xx-loc hs-ops hs-ops* **by** *unfold-locales*
interpretation *hs-to-fifo*: *set-to-list-loc hs-ops fifo-ops* **by** *unfold-locales*

setup *Locale-Code.close-block*

5.2 Generating Indices of Rules

Rule indices are pieces of extra information that may be attached to a tree automaton. There are three possible rule indices

f index of rules by function symbol

s index of rules by lhs

sf index of rules

definition *build-rule-index*
:: (('*q*,'*l*) *ta-rule* ⇒ '*i*::*hashable*) ⇒ ('*q*,'*l*) *ta-rule ls*
⇒ ('*i*,('*q*,'*l*) *ta-rule ls*) *hm*
where *build-rule-index* == *hll-idx.idx-build*

definition *build-rule-index-f* *δ* == *build-rule-index* (λ*r*. *rhsl r*) *δ*
definition *build-rule-index-s* *δ* == *build-rule-index* (λ*r*. *lhs r*) *δ*
definition *build-rule-index-sf* *δ* == *build-rule-index* (λ*r*. (*lhs r*, *rhsl r*)) *δ*

lemma *build-rule-index-f-correct*[*simp*]:
assumes *I*[*simp*, *intro*!]: *ls-invar δ*
shows *hll-idx.is-index rhsl* (*ls-α δ*) (*build-rule-index-f δ*)
apply (*unfold build-rule-index-f-def build-rule-index-def*)
apply (*simp add*: *hll-idx.idx-build-is-index*)
done

lemma *build-rule-index-s-correct*[*simp*]:
assumes *I*[*simp*, *intro*!]: *ls-invar δ*
shows
hll-idx.is-index lhs (*ls-α δ*) (*build-rule-index-s δ*)
by (*unfold build-rule-index-s-def build-rule-index-def*)
(*simp add*: *hll-idx.idx-build-is-index*)

lemma *build-rule-index-sf-correct*[*simp*]:
assumes *I*[*simp*, *intro*!]: *ls-invar δ*
shows
hll-idx.is-index (λ*r*. (*lhs r*, *rhsl r*)) (*ls-α δ*) (*build-rule-index-sf δ*)
by (*unfold build-rule-index-sf-def build-rule-index-def*)
(*simp add*: *hll-idx.idx-build-is-index*)

5.3 Tree Automaton with Optional Indices

A tree automaton contains a hashset of initial states, a list-set of rules and several (optional) rule indices.

record (**overloaded**) ('*q*,'*l*) *hashedTa* =

```
  — Initial states
  hta-Qi :: 'q hs
  — Rules
  hta-δ :: ('q,'l) ta-rule ls
  — Rules by function symbol
  hta-idx-f :: ('l,('q,'l) ta-rule ls) hm option
  — Rules by lhs state
  hta-idx-s :: ('q,('q,'l) ta-rule ls) hm option
  — Rules by lhs state and function symbol
  hta-idx-sf :: ('q×'l,('q,'l) ta-rule ls) hm option

  — Abstraction of a concrete tree automaton to an abstract one
definition hta-α
  where hta-α H = (| ta-initial = hs-α (hta-Qi H), ta-rules = ls-α (hta-δ H) |)

  — Builds the f-index if not present
definition hta-ensure-idx-f H ==
  case hta-idx-f H of
    None ⇒ H(| hta-idx-f := Some (build-rule-index-f (hta-δ H)) |) |
    Some - ⇒ H


  — Builds the s-index if not present
definition hta-ensure-idx-s H ==
  case hta-idx-s H of
    None ⇒ H(| hta-idx-s := Some (build-rule-index-s (hta-δ H)) |) |
    Some - ⇒ H


  — Builds the sf-index if not present
definition hta-ensure-idx-sf H ==
  case hta-idx-sf H of
    None ⇒ H(| hta-idx-sf := Some (build-rule-index-sf (hta-δ H)) |) |
    Some - ⇒ H


lemma hta-ensure-idx-f-correct-α[simp]:
  hta-α (hta-ensure-idx-f H) = hta-α H
  by (simp add: hta-ensure-idx-f-def hta-α-def split: option.split)
lemma hta-ensure-idx-s-correct-α[simp]:
  hta-α (hta-ensure-idx-s H) = hta-α H
  by (simp add: hta-ensure-idx-s-def hta-α-def split: option.split)
lemma hta-ensure-idx-sf-correct-α[simp]:
  hta-α (hta-ensure-idx-sf H) = hta-α H
  by (simp add: hta-ensure-idx-sf-def hta-α-def split: option.split)

lemma hta-ensure-idx-other[simp]:
  hta-Qi (hta-ensure-idx-f H) = hta-Qi H
  hta-δ (hta-ensure-idx-f H) = hta-δ H
```

```
  hta-Qi (hta-ensure-idx-s H) = hta-Qi H
  hta-δ (hta-ensure-idx-s H) = hta-δ H

  hta-Qi (hta-ensure-idx-sf H) = hta-Qi H
  hta-δ (hta-ensure-idx-sf H) = hta-δ H
  by (auto
    simp add: hta-ensure-idx-f-def hta-ensure-idx-s-def hta-ensure-idx-sf-def
    split: option.split)

  — Check whether the f-index is present
definition hta-has-idx-f H == hta-idx-f H ≠ None
  — Check whether the s-index is present
definition hta-has-idx-s H == hta-idx-s H ≠ None
  — Check whether the sf-index is present
definition hta-has-idx-sf H == hta-idx-sf H ≠ None

lemma hta-idx-f-pres
  [simp, intro!]: hta-has-idx-f (hta-ensure-idx-f H) and
  [simp, intro]: hta-has-idx-s H ⟹ hta-has-idx-s (hta-ensure-idx-f H) and
  [simp, intro]: hta-has-idx-sf H ⟹ hta-has-idx-sf (hta-ensure-idx-f H)
  by (simp-all
    add: hta-has-idx-f-def hta-has-idx-s-def hta-has-idx-sf-def
         hta-ensure-idx-f-def
    split: option.split)

lemma hta-idx-s-pres
  [simp, intro!]: hta-has-idx-s (hta-ensure-idx-s H) and
  [simp, intro]: hta-has-idx-f H ⟹ hta-has-idx-f (hta-ensure-idx-s H) and
  [simp, intro]: hta-has-idx-sf H ⟹ hta-has-idx-sf (hta-ensure-idx-s H)
  by (simp-all
    add: hta-has-idx-f-def hta-has-idx-s-def hta-has-idx-sf-def
         hta-ensure-idx-s-def
    split: option.split)

lemma hta-idx-sf-pres
  [simp, intro!]: hta-has-idx-sf (hta-ensure-idx-sf H) and
  [simp, intro]: hta-has-idx-f H ⟹ hta-has-idx-f (hta-ensure-idx-sf H) and
  [simp, intro]: hta-has-idx-s H ⟹ hta-has-idx-s (hta-ensure-idx-sf H)
  by (simp-all
    add: hta-has-idx-f-def hta-has-idx-s-def hta-has-idx-sf-def
         hta-ensure-idx-sf-def
    split: option.split)
```

The lookup functions are only defined if the required index is present. This enforces generation of the index before applying lookup functions.

```
  — Lookup rules by function symbol
definition hta-lookup-f f H == hll-idx.lookup f (the (hta-idx-f H))
  — Lookup rules by lhs-state
```

definition *hta-lookup-s q H == hll-idx.lookup q (the (hta-idx-s H))*
— Lookup rules by function symbol and lhs-state
definition *hta-lookup-sf q f H == hll-idx.lookup (q,f) (the (hta-idx-sf H))*

— This locale defines the invariants of a tree automaton
locale *hashedTa* =
fixes *H* :: (*'Q*::*hashable*,*'L*::*hashable*) *hashedTa*

— The involved sets satisfy their invariants
assumes *invar*[*simp*, *intro*!]:
hs-invar (*hta-Qi H*)
ls-invar (*hta-δ H*)

— The indices are correct, if present
assumes *index-correct*:
hta-idx-f H = Some idx-f
⟹ *hll-idx.is-index rhsl* (*ls-α* (*hta-δ H*)) *idx-f*
hta-idx-s H = Some idx-s
⟹ *hll-idx.is-index lhs* (*ls-α* (*hta-δ H*)) *idx-s*
hta-idx-sf H = Some idx-sf
⟹ *hll-idx.is-index* (λ*r*. (*lhs r*, *rhsl r*)) (*ls-α* (*hta-δ H*)) *idx-sf*

begin
— Inside this locale, some shorthand notations for the sets of rules and initial states are used
abbreviation *δ == hta-δ H*
abbreviation *Qi == hta-Qi H*

— The lookup-xxx operations are correct
lemma *hta-lookup-f-correct*:
hta-has-idx-f H ⟹ *ls-α* (*hta-lookup-f f H*) = {*r*∈*ls-α δ* . *rhsl r* = *f*}
hta-has-idx-f H ⟹ *ls-invar* (*hta-lookup-f f H*)
apply (*cases hta-has-idx-f H*)
apply (*unfold hta-has-idx-f-def hta-lookup-f-def*)
apply (*auto*
simp add: *hll-idx.lookup-correct*[*OF index-correct*(*1*)]
index-def)
done

lemma *hta-lookup-s-correct*:
hta-has-idx-s H ⟹ *ls-α* (*hta-lookup-s q H*) = {*r*∈*ls-α δ* . *lhs r* = *q*}
hta-has-idx-s H ⟹ *ls-invar* (*hta-lookup-s q H*)
apply (*cases hta-has-idx-s H*)
apply (*unfold hta-has-idx-s-def hta-lookup-s-def*)
apply (*auto*
simp add: *hll-idx.lookup-correct*[*OF index-correct*(*2*)]
index-def)
done

lemma *hta-lookup-sf-correct*:
hta-has-idx-sf H
$\Longrightarrow$ *ls-α* (*hta-lookup-sf q f H*) = {*r*∈*ls-α δ* . *lhs r* = *q* ∧ *rhsl r* = *f*}
hta-has-idx-sf H $\Longrightarrow$ *ls-invar* (*hta-lookup-sf q f H*)
apply (*cases hta-has-idx-sf H*)
apply (*unfold hta-has-idx-sf-def hta-lookup-sf-def*)
apply (*auto*
simp add: *hll-idx.lookup-correct*[*OF index-correct*(*3*)]
index-def)
done

— The ensure-index operations preserve the invariants
lemma *hta-ensure-idx-f-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-f H*)
apply (*unfold-locales*)
apply (*auto*)
apply (*auto*
simp add: *hta-ensure-idx-f-def hta-ensure-idx-s-def hta-ensure-idx-sf-def*
index-correct
split: *option.split-asm*)
done

lemma *hta-ensure-idx-s-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-s H*)
apply (*unfold-locales*)
apply (*auto*)
apply (*auto*
simp add: *hta-ensure-idx-f-def hta-ensure-idx-s-def hta-ensure-idx-sf-def*
index-correct
split: *option.split-asm*)
done

lemma *hta-ensure-idx-sf-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-sf H*)
apply (*unfold-locales*)
apply (*auto*)
apply (*auto*
simp add: *hta-ensure-idx-f-def hta-ensure-idx-s-def hta-ensure-idx-sf-def*
index-correct
split: *option.split-asm*)
done

The abstract tree automaton satisfies the invariants for an abstract tree automaton

lemma *hta-α-is-ta*[*simp*, *intro*!]: *tree-automaton* (*hta-α H*)
apply *unfold-locales*
apply (*unfold hta-α-def*)
apply *auto*
done

end

— Add some lemmas to simpset – also outside the locale
lemmas [*simp*, *intro*] =
hashedTa.hta-ensure-idx-f-correct
hashedTa.hta-ensure-idx-s-correct
hashedTa.hta-ensure-idx-sf-correct

— Build a tree automaton from a set of initial states and a set of rules
definition *init-hta Qi δ* ==
(‖ *hta-Qi*= *Qi*,
hta-δ = *δ*,
hta-idx-f = *None*,
hta-idx-s = *None*,
hta-idx-sf = *None*
‖)

— Building a tree automaton from a valid tree automaton yields again a valid tree automaton. This operation has the only effect of removing the indices.
lemma (**in** *hashedTa*) *init-hta-is-hta*:
hashedTa (*init-hta* (*hta-Qi H*) (*hta-δ H*))
apply (*unfold-locales*)
apply (*unfold init-hta-def*)
apply (*auto*)
done

5.4 Algorithm for the Word Problem

lemma *r-match-by-laz*: *r-match L l* = *list-all-zip* (*λQ q*. *q*∈*Q*) *L l*
by (*unfold r-match-alt list-all-zip-alt*)
auto

Executable function that computes the set of accepting states for a given tree

fun *faccs′* **where**
faccs′ H (*NODE f ts*) = (
let Qs = *List.map* (*faccs′ H*) *ts in*
ll-set-xy.g-image-filter (*λr*. *case r of* (*q* → *f′ qs*) ⇒
if list-all-zip (*λQ q*. *ls-memb q Q*) *Qs qs then Some* (*lhs r*) *else None*
)
(*hta-lookup-f f H*)
)

— Executable algorithm to decide the word-problem. The first version depends on the f-index to be present, the second version computes the index if not present.
definition *hta-mem′ t H* == ¬*lh-set-xx.g-disjoint* (*faccs′ H t*) (*hta-Qi H*)
definition *hta-mem t H* == *hta-mem′ t* (*hta-ensure-idx-f H*)

context *hashedTa*

begin

lemma *faccs'-invar*:
assumes *HI*[*simp*, *intro*!]: *hta-has-idx-f H*
shows *ls-invar* (*faccs' H t*) (**is** *?T1*)
list-all ls-invar (*List.map* (*faccs' H*) *ts*) (**is** *?T2*)
proof −
have *?T1* ∧ *?T2*
apply (*induct rule*: *compat-tree-tree-list.induct*)
apply (*auto simp add*: *ll-set-xy.image-filter-correct hta-lookup-f-correct*)
done
thus *?T1 ?T2* **by** *auto*
qed

declare *faccs'-invar*(*1*)[*simp*, *intro*]

lemma *faccs'-correct*:
assumes *HI*[*simp*, *intro*!]: *hta-has-idx-f H*
shows
ls-α (*faccs' H t*) = *faccs* (*ls-α* (*hta-δ H*)) *t* (**is** *?T1*)
List.map ls-α (*List.map* (*faccs' H*) *ts*)
= *List.map* (*faccs* (*ls-α* (*hta-δ H*))) *ts* (**is** *?T2*)
proof −
have *?T1* ∧ *?T2*
proof (*induct rule*: *compat-tree-tree-list.induct*)
case (*NODE f ts*)
let *?δ* = (*ls-α* (*hta-δ H*))
have *faccs ?δ* (*NODE f ts*) = (
let Qs = *List.map* (*faccs ?δ*) *ts in*
{*q*. ∃ *r*∈*?δ*. *r-matchc q f Qs r* })
by (*rule faccs.simps*)
also note *NODE.hyps*[*symmetric*]
finally have
1: *faccs ?δ* (*NODE f ts*)
= (*let Qs* = *List.map ls-α* (*List.map* (*faccs' H*) *ts*) *in*
{*q*. ∃ *r*∈*?δ*. *r-matchc q f Qs r* }) **.**
{
fix *Qsc*:: *'Q ls list*
assume *QI*: *list-all ls-invar Qsc*
let *?Qs* = *List.map ls-α Qsc*
have { *q*. ∃ *r*∈*?δ*. *r-matchc q f ?Qs r* }
= { *q*. ∃ *qs*. (*q* → *f qs*)∈*?δ* ∧ *r-match ?Qs qs* }
apply (*safe*)
apply (*case-tac r*)
apply *auto* [*1*]
apply *force*
done
also have ... = *lhs* ‘ { *r*∈{*r*∈*?δ*. *rhsl r* = *f*}.
case r of (*q* → *f' qs*) ⇒ *r-match ?Qs qs*}

apply *auto*
apply *force*
apply (*case-tac xa*)
apply *auto*
done
finally have
1: { *q*. ∃ *r*∈*?δ*. *r-matchc q f ?Qs r* }
= *lhs* ‘ { *r*∈{*r*∈*?δ*. *rhsl r* = *f*}.
case r of (*q* → *f′ qs*) ⇒ *r-match ?Qs qs*}
by *auto*
from *QI* **have**
[*simp*]: !!*qs*. *list-all-zip* (λ*Q q*. *q*∈*ls-α Q*) *Qsc qs*
⟷ *list-all-zip* (λ*Q q*. *ls-memb q Q*) *Qsc qs*
apply (*induct Qsc*)
apply (*case-tac qs*)
apply *auto* [*2*]
apply (*case-tac qs*)
apply (*auto simp add*: *ls.correct*) [*2*]
done
have *2*: !!*qs*. *r-match ?Qs qs* = *list-all-zip* (λ*a b*. *ls-memb b a*) *Qsc qs*
apply (*unfold r-match-by-laz*)
apply (*simp add*: *list-all-zip-map1*)
done
from *1* **have**
{ *q*. ∃ *r*∈*?δ*. *r-matchc q f ?Qs r* }
= *lhs* ‘ { *r*∈{*r*∈*?δ*. *rhsl r* = *f*}.
case r of (*q* → *f′ qs*) ⇒
list-all-zip (λ*a b*. *ls-memb b a*) *Qsc qs*}
by (*simp only*: *2*)
also have
. . . = *lhs* ‘ { *r*∈*ls-α* (*hta-lookup-f f H*).
case r of (*q* → *f′ qs*) ⇒
list-all-zip (λ*a b*. *ls-memb b a*) *Qsc qs*}
by (*simp add*: *hta-lookup-f-correct*)
also have
. . . = *ls-α* (*ll-set-xy.g-image-filter*
(λ*r*. *case r of* (*q* → *f′ qs*) ⇒
(*if* (*list-all-zip* (λ*Q q*. *ls-memb q Q*) *Qsc qs*) *then Some* (*lhs r*) *else None*))
(*hta-lookup-f f H*)
)
apply (*simp add*: *ll-set-xy.image-filter-correct hta-lookup-f-correct*)
apply (*auto split*: *ta-rule.split*)
apply (*rule-tac x*=*xa* **in** *exI*)
apply *auto*
apply (*case-tac a*)
apply (*simp add*: *image-iff*)
apply (*rule-tac x*=*a* **in** *exI*)
apply *auto*

```
      done
    finally have
      { q. ∃r∈?δ. r-matchc q f ?Qs r }
       = ls-α ( ll-set-xy.g-image-filter
                 (λr. case r of (q → f' qs) ⇒
                  (if (list-all-zip (λQ q. ls-memb q Q) Qsc qs) then Some (lhs r)
else None))
                 (hta-lookup-f f H)) .
  } note 2=this

  from
    1
    2[ where Qsc2 = (List.map (faccs' H) ts),
      simplified faccs'-invar[OF HI]]
  show ?case by simp
 qed simp-all
 thus ?T1 ?T2 by auto
qed

 — Correctness of the algorithms for the word problem
lemma hta-mem'-correct:
 hta-has-idx-f H ⟹ hta-mem' t H ⟷ t∈ta-lang (hta-α H)
 apply (unfold ta-lang-def hta-α-def hta-mem'-def)
 apply (auto simp add: lh-set-xx.disjoint-correct faccs'-correct faccs-alt)
 done

theorem hta-mem-correct: hta-mem t H ⟷ t∈ta-lang (hta-α H)
 using hashedTa.hta-mem'-correct[OF hta-ensure-idx-f-correct, simplified]
 apply (unfold hta-mem-def)
 apply simp
 done

end
```

5.5 Product Automaton and Intersection

5.5.1 Brute Force Product Automaton

In this section, an algorithm that computes the product automaton without reduction is implemented. While the runtime is always quadratic, this algorithm is very simple and the constant factors are smaller than that of the version with integrated reduction. Moreover, lazy languages like Haskell seem to profit from this algorithm.

```
definition δ-prod-h
 :: ('q1::hashable,'l::hashable) ta-rule ls
    ⇒ ('q2::hashable,'l) ta-rule ls ⇒ ('q1×'q2,'l) ta-rule ls
 where δ-prod-h δ1 δ2 ==
  lll-iflt-cp.inj-image-filter-cp (λ(r1,r2). r-prod r1 r2)
            (λ(r1,r2). rhsl r1 = rhsl r2
```

$\wedge$ *length* (*rhsq r1*) = *length* (*rhsq r2*))
δ1 δ2

lemma *r-prod-inj*:
⟦ *rhsl r1* = *rhsl r2*; *length* (*rhsq r1*) = *length* (*rhsq r2*);
rhsl r1′ = *rhsl r2′*; *length* (*rhsq r1′*) = *length* (*rhsq r2′*);
r-prod r1 r2 = *r-prod r1′ r2′* ⟧ ⟹ *r1*=*r1′* ∧ *r2*=*r2′*
apply (*cases r1*, *cases r2*, *cases r1′*, *cases r2′*)
apply (*auto dest*: *zip-inj*)
done

lemma *δ-prod-h-correct*:
assumes *INV*[*simp*]: *ls-invar δ1* *ls-invar δ2*
shows
ls-α (*δ-prod-h δ1 δ2*) = *δ-prod* (*ls-α δ1*) (*ls-α δ2*)
ls-invar (*δ-prod-h δ1 δ2*)
apply (*unfold δ-prod-def δ-prod-h-def*)
apply (*subst lll-iflt-cp.inj-image-filter-cp-correct*)
apply *simp-all* [*2*]
using *r-prod-inj*
apply (*auto intro*!: *inj-onI*) []
apply *auto* []
apply (*case-tac xa*, *case-tac y*, *simp*, *blast*)
apply *force*
apply *simp*
done

definition *hta-prodWR H1 H2* ==
init-hta (*hhh-cart.cart* (*hta-Qi H1*) (*hta-Qi H2*)) (*δ-prod-h* (*hta-δ H1*) (*hta-δ H2*))

lemma *hta-prodWR-correct-aux*:
assumes *A*: *hashedTa H1* *hashedTa H2*
shows
hta-α (*hta-prodWR H1 H2*) = *ta-prod* (*hta-α H1*) (*hta-α H2*) (**is** *?T1*)
hashedTa (*hta-prodWR H1 H2*) (**is** *?T2*)
proof −
interpret *a1*: *hashedTa H1* + *a2*: *hashedTa H2* **using** *A* **.**
show *?T1 ?T2*
apply (*unfold hta-prodWR-def init-hta-def hta-α-def ta-prod-def*)
apply (*simp add*: *hhh-cart.cart-correct δ-prod-h-correct*)
apply (*unfold-locales*)
apply (*simp-all add*: *hhh-cart.cart-correct δ-prod-h-correct*)
done
qed

lemma *hta-prodWR-correct*:
assumes *TA*: *hashedTa H1* *hashedTa H2*
shows

```
    ta-lang (hta-α (hta-prodWR H1 H2))
     = ta-lang (hta-α H1) ∩ ta-lang (hta-α H2)
    hashedTa (hta-prodWR H1 H2)
  by (simp-all add: hta-prodWR-correct-aux[OF TA] ta-prod-correct-aux1)
```

5.5.2 Product Automaton with Forward-Reduction

A more elaborated algorithm combines forward-reduction and the product construction, i.e. product rules are only created ,,by need".

```
  — State of the product-automaton DFS-algorithm
type-synonym ('q1,'q2,'l) pa-state
  = ('q1×'q2) hs × ('q1×'q2) list × ('q1×'q2,'l) ta-rule ls

  — Abstraction mapping to algorithm specified in Section 4.
definition pa-α
  :: ('q1::hashable,'q2::hashable,'l::hashable) pa-state
     ⇒ ('q1,'q2,'l) frp-state
  where pa-α S == let (Q,W,δd)=S in (hs-α Q,W,ls-α δd)

definition pa-cond
  :: ('q1::hashable,'q2::hashable,'l::hashable) pa-state ⇒ bool
  where pa-cond S == let (Q,W,δd) = S in W≠[]

  — Adds all successor states to the set of discovered states and to the worklist
fun pa-upd-rule
  :: ('q1×'q2) hs ⇒ ('q1×'q2) list
     ⇒ (('q1::hashable)×('q2::hashable)) list
     ⇒ (('q1×'q2) hs × ('q1×'q2) list)
  where
  pa-upd-rule Q W [] = (Q,W) |
  pa-upd-rule Q W (qp#qs) = (
    if ¬ hs-memb qp Q then
      pa-upd-rule (hs-ins qp Q) (qp# W) qs
    else pa-upd-rule Q W qs
  )


definition pa-step
  :: ('q1::hashable,'l::hashable) hashedTa
     ⇒ ('q2::hashable,'l) hashedTa
     ⇒ ('q1,'q2,'l) pa-state ⇒ ('q1,'q2,'l) pa-state
  where pa-step H1 H2 S == let
    (Q,W,δd)=S;
    (q1,q2)=hd W
  in
    ls-iteratei (hta-lookup-s q1 H1) (λ-. True) (λr1 res.
      ls-iteratei (hta-lookup-sf q2 (rhsl r1) H2) (λ-. True) (λr2 res.
        if (length (rhsq r1) = length (rhsq r2)) then
          let
```

```
        rp=r-prod r1 r2;
        (Q,W,δd) = res;
        (Q',W') = pa-upd-rule Q W (rhsq rp)
      in
        (Q',W',ls-ins-dj rp δd)
    else
      res
  ) res
) (Q,tl W,δd)


definition pa-initial
  :: ('q1::hashable,'l::hashable) hashedTa
     ⇒ ('q2::hashable,'l) hashedTa
     ⇒ ('q1,'q2,'l) pa-state
where pa-initial H1 H2 ==
  let Qip = hhh-cart.cart (hta-Qi H1) (hta-Qi H2) in (
    Qip,
    hs-to-list Qip,
    ls-empty ()
  )

definition pa-invar-add::
  ('q1::hashable,'q2::hashable,'l::hashable) pa-state set
  where pa-invar-add == { (Q,W,δd). hs-invar Q ∧ ls-invar δd }

definition pa-invar H1 H2 ==
  pa-invar-add ∩ {s. (pa-α s) ∈ frp-invar (hta-α H1) (hta-α H2)}

definition pa-det-algo H1 H2
  == (| dwa-cond=pa-cond,
       dwa-step = pa-step H1 H2,
       dwa-initial = pa-initial H1 H2,
       dwa-invar = pa-invar H1 H2 |)

lemma pa-upd-rule-correct:
  assumes INV[simp, intro!]: hs-invar Q
  assumes FMT: pa-upd-rule Q W qs = (Q',W')
  shows
    hs-invar Q' (is ?T1)
    hs-α Q' = hs-α Q ∪ set qs (is ?T2)
    ∃ Wn. distinct Wn ∧ set Wn = set qs − hs-α Q ∧ W'=Wn@W (is ?T3)
proof −
  from INV FMT have ?T1 ∧ ?T2 ∧ ?T3
  proof (induct qs arbitrary: Q W Q' W')
    case Nil thus ?case by simp
  next
    case (Cons q qs Q W Q' W')
    show ?case
```

```
    proof (cases q∈hs-α Q)
      case True
      obtain Qh Wh where RF: pa-upd-rule Q W qs = (Qh,Wh) by force
      with True Cons.prems have [simp]: Q'=Qh    W'=Wh
        by (auto simp add: hs.correct)
      from Cons.hyps[OF Cons.prems(1) RF] have
        hs-invar Qh
        hs-α Qh = hs-α Q ∪ set qs
        (∃ Wn. distinct Wn ∧ set Wn = set qs − hs-α Q ∧ Wh = Wn @ W)
        by auto
      thus ?thesis using True by auto
    next
      case False
      with Cons.prems have RF: pa-upd-rule (hs-ins q Q) (q#W) qs = (Q',W')
        by (auto simp add: hs.correct)

      from Cons.hyps[OF - RF] Cons.prems(1) have
        hs-invar Q'
        hs-α Q' = insert q (hs-α Q) ∪ set (qs)
        ∃ Wn. distinct Wn
              ∧ set Wn = set qs − insert q (hs-α Q)
              ∧ W' = Wn @ q # W
        by (auto simp add: hs.correct)
      thus ?thesis using False by auto
    qed
  qed
  thus ?T1 ?T2 ?T3 by auto
qed


lemma pa-step-correct:
  assumes TA: hashedTa H1    hashedTa H2
  assumes idx[simp]: hta-has-idx-s H1    hta-has-idx-sf H2
  assumes INV: (Q,W,δd)∈pa-invar H1 H2
  assumes COND: pa-cond (Q,W,δd)
  shows
    (pa-step H1 H2 (Q,W,δd))∈pa-invar-add (is ?T1)
    (pa-α (Q,W,δd), pa-α (pa-step H1 H2 (Q,W,δd)))
     ∈ frp-step (ls-α (hta-δ H1)) (ls-α (hta-δ H2)) (is ?T2)
proof −
  interpret h1: hashedTa H1 by fact
  interpret h2: hashedTa H2 by fact

  from COND obtain q1 q2 Wtl where
    [simp]: W=(q1,q2)#Wtl
    by (cases W) (auto simp add: pa-cond-def)

  from INV have [simp]: hs-invar Q    ls-invar δd
    by (auto simp add: pa-invar-add-def pa-invar-def)
```

```
define inv where inv = (λδp (Q', W', δd').
  hs-invar Q'
  ∧ ls-invar δd'
  ∧ (∃ Wn. distinct Wn
        ∧ set Wn = (f-succ δp `` {(q1,q2)}) − hs-α Q
        ∧ W'=Wn@Wtl
        ∧ hs-α Q'=hs-α Q ∪ (f-succ δp `` {(q1,q2)}))
  ∧ (ls-α δd' = ls-α δd ∪ {r∈δp. lhs r = (q1,q2) }))

have G: inv (δ-prod (ls-α (hta-δ H1)) (ls-α (hta-δ H2)))
           (pa-step H1 H2 (Q,W,δd))
  apply (unfold pa-step-def)
  apply simp
  apply (rule-tac
    I=λit1 res. inv (δ-prod (ls-α (hta-δ H1) − it1) (ls-α (hta-δ H2))) res
    in ls.iterate-rule-P)
  — Invar
  apply (simp add: h1.hta-lookup-s-correct)
  — Initial
  apply (fastforce simp add: inv-def δ-prod-def h1.hta-lookup-s-correct
                             f-succ-alt)
  — Step
  apply (rule-tac
    I=λit2 res. inv ( δ-prod (ls-α (hta-δ H1) − it) (ls-α (hta-δ H2))
                    ∪ δ-prod {x} (ls-α (hta-δ H2) − it2))
                  res
    in ls.iterate-rule-P)
    — Invar
    apply (simp add: h2.hta-lookup-sf-correct)
    — Init
    apply (case-tac σ)
    apply (simp add: inv-def h1.hta-lookup-s-correct h2.hta-lookup-sf-correct)
    apply (force simp add: f-succ-alt elim: δ-prodE intro: δ-prodI) [1]
    — Step
    defer — Requires considerably more work: Deferred to Isar proof below
    — Final
    apply (simp add: h1.hta-lookup-s-correct h2.hta-lookup-sf-correct)
    apply (auto) [1]
    apply (subgoal-tac
      ls-α (hta-δ H1) − (it − {x}) = (ls-α (hta-δ H1) − it) ∪ {x})
    apply (simp add: δ-prod-insert)
    apply (subst Un-commute)
    apply simp
    apply blast
  — Final
  apply force
proof goal-cases
  case prems: (1 r1 it1 resxh r2 it2 resh)
```

— Resolve lookup-operations
hence G':
$it1 \subseteq \{r \in ls\text{-}\alpha\ (hta\text{-}\delta\ H1).\ lhs\ r = q1\}$
$it2 \subseteq \{r \in ls\text{-}\alpha\ (hta\text{-}\delta\ H2).\ lhs\ r = q2 \wedge rhsl\ r = rhsl\ r1\}$
by (*simp-all add*: *h1.hta-lookup-s-correct h2.hta-lookup-sf-correct*)

— Basic reasoning setup
from *prems*(1,4) G' **have**
[*simp*]: $ls\text{-}\alpha\ (hta\text{-}\delta\ H2) - (it2 - \{r2\}) = (ls\text{-}\alpha\ (hta\text{-}\delta\ H2) - it2) \cup \{r2\}$
by *auto*
obtain $Qh\ Wh\ \delta dh\ Q'\ W'\ \delta d'$ **where** [*simp*]: $resh=(Qh,Wh,\delta dh)$
by (*cases resh*) *fastforce*
from *prems*(6) **have** *INVAH*[*simp*]: $hs\text{-}invar\ Qh \quad ls\text{-}invar\ \delta dh$
by (*auto simp add*: *inv-def*)

— The involved rules have the same label, and their lhs is determined
from *prems*(1,4) G' **obtain** $l\ qs1\ qs2$ **where**
RULE-FMT: $r1 = (q1 \to l\ qs1) \quad r2=(q2 \to l\ qs2)$
apply (*cases r1*, *cases r2*)
apply *force*
done

{
— If the rhs have different lengths, the algorithm ignores the rule:
assume *LEN*: $length\ (rhsq\ r1) \neq length\ (rhsq\ r2)$

hence [*simp*]: $\delta\text{-}prod\text{-}sng2\ \{r1\}\ r2 = \{\}$
by (*auto simp add*: *δ-prod-sng2-def split*: *ta-rule.split*)

have *?case* **using** *prems*
by (*simp add*: *LEN δ-prod-insert*)
} moreover {
— If the rhs have the same length, the rule is inserted
assume *LEN*: $length\ (rhsq\ r1) = length\ (rhsq\ r2)$
hence [*simp*]: $length\ qs1 = length\ qs2$ **by** (*simp add*: *RULE-FMT*)

hence [*simp*]: $\delta\text{-}prod\text{-}sng2\ \{r1\}\ r2 = \{(q1,q2) \to l\ (zip\ qs1\ qs2)\}$
using *prems*(1,4) G'
by (*auto simp add*: *δ-prod-sng2-def RULE-FMT*)

— Obtain invariant of previous state
from *prems*(6)[*unfolded inv-def*, *simplified*] **obtain** *Wn* **where** *INVH*:
$distinct\ Wn$
$set\ Wn = f\text{-}succ\ (\delta\text{-}prod\ (ls\text{-}\alpha\ (hta\text{-}\delta\ H1) - it1)\ (ls\text{-}\alpha\ (hta\text{-}\delta\ H2))$
$\cup\ \delta\text{-}prod\ \{r1\}\ (ls\text{-}\alpha\ (hta\text{-}\delta\ H2) - it2))$
$``\ \{(q1,\ q2)\} - hs\text{-}\alpha\ Q$
$Wh = Wn\ @\ Wtl$
$hs\text{-}\alpha\ Qh = hs\text{-}\alpha\ Q$
$\cup\ f\text{-}succ\ (\delta\text{-}prod\ (ls\text{-}\alpha\ (hta\text{-}\delta\ H1) - it1)\ (ls\text{-}\alpha\ (hta\text{-}\delta\ H2))$

```
                    ∪ δ-prod {r1} (ls-α (hta-δ H2) − it2))
              `` {(q1, q2)}
  ls-α δdh = ls-α δd
    ∪ {r. ( r ∈ δ-prod (ls-α (hta-δ H1) − it1) (ls-α (hta-δ H2))
          ∨ r ∈ δ-prod {r1} (ls-α (hta-δ H2) − it2)
        ) ∧ lhs r = (q1, q2)
      }
  by blast

— Required to justify disjoint insert
have RPD: r-prod r1 r2 ∉ ls-α δdh
proof −
  from INV[unfolded pa-invar-def frp-invar-def frp-invar-add-def]
  have LSDD:
    ls-α δd = {r ∈ δ-prod (ls-α (hta-δ H1)) (ls-α (hta-δ H2)).
                lhs r ∈ hs-α Q − set W}
    by (auto simp add: pa-α-def hta-α-def)
  have r-prod r1 r2 ∉ ls-α δd
  proof
    assume r-prod r1 r2 ∈ ls-α δd
    with LSDD have lhs (r-prod r1 r2) ∉ set W by auto
    moreover from prems(1,4) G' have lhs (r-prod r1 r2) = (q1,q2)
      by (cases r1, cases r2) auto
    ultimately show False by simp
  qed
  moreover from prems(6) have ls-α δdh =
    ls-α δd ∪
    {r. ( r ∈ δ-prod (ls-α (hta-δ H1) − it1) (ls-α (hta-δ H2))
          ∨ r ∈ δ-prod {r1} (ls-α (hta-δ H2) − it2)
        ) ∧ lhs r = (q1, q2)} (is -= - ∪ ?s)
    by (simp add: inv-def)
  moreover have r-prod r1 r2 ∉ ?s using prems(1,4) G'(2) LEN
    apply (cases r1, cases r2)
    apply (auto simp add: δ-prod-def)
    done
  ultimately show ?thesis by blast
qed

— Correctness of result of pa-upd-rule
obtain Q' W' where
  PAUF: (pa-upd-rule Qh Wh (rhsq (r-prod r1 r2))) = (Q',W')
  by force
from pa-upd-rule-correct[OF INVAH(1) PAUF] obtain Wnn where UC:
  hs-invar Q'
  hs-α Q' = hs-α Qh ∪ set (rhsq (r-prod r1 r2))
  distinct Wnn
  set Wnn = set (rhsq (r-prod r1 r2)) − hs-α Qh
  W' = Wnn @ Wh
  by blast
```

```
    — Put it all together
    have ?case
      apply (simp add: LEN Let-def ls.ins-dj-correct[OF INVAH(2) RPD]
                       PAUF inv-def UC(1))
      apply (intro conjI)
      apply (rule-tac x=Wnn@Wn in exI)
      apply (auto simp add: f-succ-alt δ-prod-insert RULE-FMT UC INVH
                            δ-prod-sng2-def δ-prod-sng1-def)
      done
  } ultimately show ?case by blast
 qed
 from G show ?T1
  by (cases pa-step H1 H2 (Q,W,δd))
     (simp add: pa-invar-add-def inv-def)
 from G show ?T2
  by (cases pa-step H1 H2 (Q,W,δd))
     (auto simp add: inv-def pa-α-def Let-def intro: frp-step.intros)

qed



 — The product-automaton algorithm is a precise implementation of its specification
lemma pa-pref-frp:
 assumes TA: hashedTa H1    hashedTa H2
 assumes idx[simp]: hta-has-idx-s H1    hta-has-idx-sf H2

 shows wa-precise-refine (det-wa-wa (pa-det-algo H1 H2))
                         (frp-algo (hta-α H1) (hta-α H2))
                         pa-α
proof −
 interpret h1: hashedTa H1 by fact
 interpret h2: hashedTa H2 by fact

 show ?thesis
  apply (unfold-locales)
  apply (auto simp add: det-wa-wa-def pa-det-algo-def pa-α-def
                        pa-cond-def frp-algo-def frp-cond-def) [1]
  apply (auto simp add: det-wa-wa-def pa-det-algo-def pa-cond-def
                        hta-α-def frp-algo-def frp-cond-def
              intro!: pa-step-correct(2)[OF TA]) [1]
  apply (auto simp add: det-wa-wa-def pa-det-algo-def pa-α-def
                        hta-α-def pa-cond-def frp-algo-def frp-cond-def
                        pa-invar-def pa-step-def pa-initial-def
                        hs.correct ls.correct Let-def hhh-cart.cart-correct
              intro: frp-initial.intros
  ) [3]
  done
```

qed

— The product automaton algorithm is a correct while-algorithm
lemma *pa-while-algo*:
assumes *TA*: *hashedTa H1* *hashedTa H2*
assumes *idx*[*simp*]: *hta-has-idx-s H1* *hta-has-idx-sf H2*

shows *while-algo* (*det-wa-wa* (*pa-det-algo H1 H2*))
proof −
interpret *h1*: *hashedTa H1* **by** *fact*
interpret *h2*: *hashedTa H2* **by** *fact*

interpret *ref*: *wa-precise-refine* (*det-wa-wa* (*pa-det-algo H1 H2*))
(*frp-algo* (*hta-α H1*) (*hta-α H2*))
pa-α
using *pa-pref-frp*[*OF TA idx*] **.**
show *?thesis*
apply (*rule ref.wa-intro*)
apply (*simp add*: *frp-while-algo*)
apply (*simp add*: *det-wa-wa-def pa-det-algo-def pa-invar-def frp-algo-def*)

apply (*auto simp add*: *det-wa-wa-def pa-det-algo-def*) [*1*]
apply (*rule pa-step-correct*(*1*)[*OF TA idx*])
apply (*auto simp add*: *pa-invar-def frp-algo-def*) [*2*]

apply (*simp add*: *det-wa-wa-def pa-det-algo-def pa-initial-def*
pa-invar-add-def Let-def hhh-cart.cart-correct ls.correct)
done
qed

— By definition, the product automaton algorithm is deterministic
lemmas *pa-det-while-algo* = *det-while-algo-intro*[*OF pa-while-algo*]

— Transferred correctness lemma
lemmas *pa-inv-final* =
wa-precise-refine.transfer-correctness[*OF pa-pref-frp frp-inv-final*]

— The next two definitions specify the product-automata algorithm. The first version requires the s-index of the first and the sf-index of the second automaton to be present, while the second version computes the required indices, if necessary
definition *hta-prod′ H1 H2* ==
let (*Q*,*W*,*δd*) = *while pa-cond* (*pa-step H1 H2*) (*pa-initial H1 H2*) *in*
init-hta (*hhh-cart.cart* (*hta-Qi H1*) (*hta-Qi H2*)) *δd*

definition *hta-prod H1 H2* ==

```
  hta-prod' (hta-ensure-idx-s H1) (hta-ensure-idx-sf H2)


lemma hta-prod'-correct-aux:
  assumes TA: hashedTa H1    hashedTa H2
  assumes idx: hta-has-idx-s H1    hta-has-idx-sf H2
  shows hta-α (hta-prod' H1 H2)
        = ta-fwd-reduce (ta-prod (hta-α H1) (hta-α H2)) (is ?T1)
       hashedTa (hta-prod' H1 H2) (is ?T2)
proof −
  interpret h1: hashedTa H1 by fact
  interpret h2: hashedTa H2 by fact

  interpret dwa: det-while-algo pa-det-algo H1 H2
    using pa-det-while-algo[OF TA idx] .

  have LC: while pa-cond (pa-step H1 H2) (pa-initial H1 H2) = dwa.loop
    by (unfold dwa.loop-def)
       (simp add: pa-det-algo-def)

  from dwa.while-proof'[OF pa-inv-final[OF TA idx]]
  show ?T1
    apply (unfold dwa.loop-def)
    apply (simp add: hta-prod'-def init-hta-def hta-α-def pa-det-algo-def)
    apply (cases (while pa-cond (pa-step H1 H2) (pa-initial H1 H2)))
    apply (simp add: pa-α-def hhh-cart.cart-correct hta-α-def)
    done

  show ?T2
    apply (simp add: hta-prod'-def LC)
    apply (rule dwa.while-proof)
    apply (case-tac s)
    apply (simp add: pa-det-algo-def pa-invar-add-def pa-invar-def init-hta-def)
    apply unfold-locales
    apply (simp-all add: hhh-cart.cart-correct)
    done
qed

theorem hta-prod'-correct:
  assumes TA: hashedTa H1    hashedTa H2
  assumes HI: hta-has-idx-s H1    hta-has-idx-sf H2
  shows
    ta-lang (hta-α (hta-prod' H1 H2))
     = ta-lang (hta-α H1) ∩ ta-lang (hta-α H2)

    hashedTa (hta-prod' H1 H2)
  by (simp-all add: hta-prod'-correct-aux[OF TA HI] ta-prod-correct-aux1)

lemma hta-prod-correct-aux:
```

assumes *TA*[*simp*]: *hashedTa H1* *hashedTa H2*
shows
hta-α (*hta-prod H1 H2*) = *ta-fwd-reduce* (*ta-prod* (*hta-α H1*) (*hta-α H2*))
hashedTa (*hta-prod H1 H2*)
by (*unfold hta-prod-def*)
(*simp-all add*: *hta-prod′-correct-aux*)

theorem *hta-prod-correct*:
assumes *TA*: *hashedTa H1* *hashedTa H2*
shows
ta-lang (*hta-α* (*hta-prod H1 H2*))
= *ta-lang* (*hta-α H1*) ∩ *ta-lang* (*hta-α H2*)
hashedTa (*hta-prod H1 H2*)
by (*simp-all add*: *hta-prod-correct-aux*[*OF TA*] *ta-prod-correct-aux1*)

5.6 Remap States

— Mapping the states of an automaton
definition *hta-remap*
:: ('*q*::*hashable* ⇒ '*qn*::*hashable*) ⇒ ('*q*,'*l*::*hashable*) *hashedTa*
⇒ ('*qn*,'*l*) *hashedTa*
where *hta-remap f H* ==
init-hta (*hh-set-xy.g-image f* (*hta-Qi H*))
(*ll-set-xy.g-image* (*remap-rule f*) (*hta-δ H*))

lemma (**in** *hashedTa*) *hta-remap-correct*:
shows *hta-α* (*hta-remap f H*) = *ta-remap f* (*hta-α H*)
hashedTa (*hta-remap f H*)
apply (*auto*
simp add: *hta-remap-def init-hta-def hta-α-def*
hh-set-xy.image-correct ll-set-xy.image-correct ta-remap-def)
apply (*unfold-locales*)
apply (*auto simp add*: *hh-set-xy.image-correct ll-set-xy.image-correct*)
done

5.6.1 Reindex Automaton

In this section, an algorithm for re-indexing the states of the automaton to an initial segment of the naturals is implemented. The language of the automaton is not changed by the reindexing operation.

— Set of states of a rule
fun *rule-states-l* **where**
rule-states-l (*q* → *f qs*) = *ls-ins q* (*ls.from-list qs*)

lemma *rule-states-l-correct*[*simp*]:
ls-α (*rule-states-l r*) = *rule-states r*
ls-invar (*rule-states-l r*)
by (*cases r*, *simp add*: *ls.correct*)+

```
definition hta-δ-states H
  == (llh-set-xyy.g-Union-image id (ll-set-xy.g-image-filter
      (λr. Some (rule-states-l r)) (hta-δ H)))

definition hta-states H ==
  hs-union (hta-Qi H) (hta-δ-states H)

lemma (in hashedTa) hta-δ-states-correct:
  hs-α (hta-δ-states H) = δ-states (ta-rules (hta-α H))
  hs-invar (hta-δ-states H)
proof (simp-all add: hta-α-def hta-δ-states-def, goal-cases)
  case 1
  have
    [simp]: ls-α (ll-set-xy.g-image-filter (λx. Some (rule-states-l x)) δ)
           = rule-states-l ‘ ls-α δ
    by (auto simp add: ll-set-xy.image-filter-correct)
  show ?case
    apply (simp add: δ-states-def)
    apply (subst
      llh-set-xyy.Union-image-correct[
        of (ll-set-xy.g-image-filter (λx. Some (rule-states-l x)) δ),
        simplified])
    apply (auto simp add: ll-set-xy.image-filter-correct)
    done

qed

lemma (in hashedTa) hta-states-correct:
  hs-α (hta-states H) = ta-rstates (hta-α H)
  hs-invar (hta-states H)
  by (simp-all
    add: hta-states-def ta-rstates-def hs.correct hta-δ-states-correct
         hta-α-def)

definition reindex-map H ==
  λq. the (hm-lookup q (hh-map-to-nat.map-to-nat (hta-states H)))

definition hta-reindex
  :: ('Q::hashable,'L::hashable) hashedTa ⇒ (nat,'L) hashedTa where
  hta-reindex H == hta-remap (reindex-map H) H

declare hta-reindex-def [code del]

  — This version is more efficient, as the map is only computed once
lemma [code]: hta-reindex H = (
  let mp = (hh-map-to-nat.map-to-nat (hta-states H)) in
    hta-remap (λq. the (hm-lookup q mp)) H)

  by (simp add: Let-def hta-reindex-def reindex-map-def)
```

```
lemma (in hashedTa) reindex-map-correct:
  inj-on (reindex-map H) (ta-rstates (hta-α H))
proof −
  have [simp]:
    reindex-map H = the ∘ hm-α (hh-map-to-nat.map-to-nat (hta-states H))
    by (rule ext)
       (simp add: reindex-map-def hm.correct
         hh-map-to-nat.map-to-nat-correct(4)
         hta-states-correct)

  show ?thesis
    apply (simp add: hta-states-correct(1)[symmetric])
    apply (rule inj-on-map-the)
    apply (simp-all add: hh-map-to-nat.map-to-nat-correct hta-states-correct(2))
    done
qed

theorem (in hashedTa) hta-reindex-correct:
  ta-lang (hta-α (hta-reindex H)) = ta-lang (hta-α H)
  hashedTa (hta-reindex H)
  apply (unfold hta-reindex-def)
  apply (simp-all
    add: hta-remap-correct tree-automaton.remap-lang[OF hta-α-is-ta]
         reindex-map-correct)
  done
```

5.7 Union

Computes the union of two automata

```
definition hta-union
  :: ('q1::hashable,'l::hashable) hashedTa
     ⇒ ('q2::hashable,'l) hashedTa
     ⇒ (('q1,'q2) ustate-wrapper,'l) hashedTa
  where hta-union H1 H2 ==
    init-hta (hs-union (hh-set-xy.g-image USW1 (hta-Qi H1))
                       (hh-set-xy.g-image USW2 (hta-Qi H2)))
             (ls-union-dj (ll-set-xy.g-image (remap-rule USW1) (hta-δ H1))
                          (ll-set-xy.g-image (remap-rule USW2) (hta-δ H2)))

lemma hta-union-correct':
  assumes TA: hashedTa H1    hashedTa H2
  shows hta-α (hta-union H1 H2)
        = ta-union-wrap (hta-α H1) (hta-α H2) (is ?T1)
        hashedTa (hta-union H1 H2) (is ?T2)
proof −
  interpret a1: hashedTa H1 + a2: hashedTa H2 using TA .
  show ?T1 ?T2
```

```
    apply (auto
      simp add: hta-union-def init-hta-def hta-α-def
                hs.correct ls.correct
                ll-set-xy.image-correct hh-set-xy.image-correct
                ta-remap-def ta-union-def ta-union-wrap-def)
    apply (unfold-locales)
    apply (auto
      simp add: hs.correct ls.correct)
    done
qed

theorem hta-union-correct:
  assumes TA: hashedTa H1    hashedTa H2
  shows
    ta-lang (hta-α (hta-union H1 H2))
     = ta-lang (hta-α H1) ∪ ta-lang (hta-α H2) (is ?T1)
    hashedTa (hta-union H1 H2) (is ?T2)
proof −
  interpret a1: hashedTa H1 + a2: hashedTa H2 using TA .
  show ?T1 ?T2
    by (simp-all add: hta-union-correct'[OF TA] ta-union-wrap-correct)
qed
```

5.8 Operators to Construct Tree Automata

This section defines operators that add initial states and rules to a tree automaton, and thus incrementally construct a tree automaton from the empty automaton.

```
— The empty automaton
definition hta-empty :: unit ⇒ ('q::hashable,'l::hashable) hashedTa
  where hta-empty u == init-hta (hs-empty ()) (ls-empty ())
lemma hta-empty-correct [simp, intro!]:
  shows (hta-α (hta-empty ())) = ta-empty
        hashedTa (hta-empty ())
  apply (auto
    simp add: init-hta-def hta-empty-def hta-α-def δ-states-def ta-empty-def
              hs.correct ls.correct)
  apply (unfold-locales)
  apply (auto simp add: hs.correct ls.correct)
  done

— Add an initial state to the automaton
definition hta-add-qi
  :: 'q ⇒ ('q::hashable,'l::hashable) hashedTa ⇒ ('q,'l) hashedTa
  where hta-add-qi qi H == init-hta (hs-ins qi (hta-Qi H)) (hta-δ H)

lemma (in hashedTa) hta-add-qi-correct[simp, intro!]:
  shows hta-α (hta-add-qi qi H)
        = ⦇ ta-initial = insert qi (ta-initial (hta-α H)),
```

```
        ta-rules = ta-rules (hta-α H)
      ⦈
    hashedTa (hta-add-qi qi H)
  apply (auto
    simp add: init-hta-def hta-add-qi-def hta-α-def δ-states-def
        hs.correct)
  apply (unfold-locales)
  apply (auto simp add: hs.correct)
  done

lemmas [simp, intro] = hashedTa.hta-add-qi-correct

— Add a rule to the automaton
definition hta-add-rule
  :: ('q,'l) ta-rule ⇒ ('q::hashable,'l::hashable) hashedTa
     ⇒ ('q,'l) hashedTa
  where hta-add-rule r H == init-hta (hta-Qi H) (ls-ins r (hta-δ H))

lemma (in hashedTa) hta-add-rule-correct[simp, intro!]:
  shows hta-α (hta-add-rule r H)
      = ⦇ ta-initial = ta-initial (hta-α H),
          ta-rules = insert r (ta-rules (hta-α H))
        ⦈
    hashedTa (hta-add-rule r H)
  apply (auto
    simp add: init-hta-def hta-add-rule-def hta-α-def
        δ-states-def ls.correct)
  apply (unfold-locales)
  apply (auto simp add: ls.correct)
  done

lemmas [simp, intro] = hashedTa.hta-add-rule-correct


  — Reduces an automaton to the given set of states
definition hta-reduce H Q ==
  init-hta (hs-inter Q (hta-Qi H))
        (ll-set-xy.g-image-filter
          (λr. if hs-memb (lhs r) Q ∧ list-all (λq. hs-memb q Q) (rhsq r) then
Some r else None)
          (hta-δ H))


theorem (in hashedTa) hta-reduce-correct:
  assumes INV[simp]: hs-invar Q
  shows
  hta-α (hta-reduce H Q) = ta-reduce (hta-α H) (hs-α Q) (is ?T1)
  hashedTa (hta-reduce H Q) (is ?T2)
  apply (auto
```

```
    simp add:
      hta-reduce-def ta-reduce-def hta-α-def init-hta-def
      hs.correct ls.correct

      list-all-iff
      reduce-rules-def rule-states-simp
      ll-set-xy.image-filter-correct
    split:
      ta-rule.split-asm
  ) [1]
  apply (unfold-locales)
  apply (unfold hta-reduce-def init-hta-def)
  apply (auto simp add: hs.correct ls.correct)
  done
```

5.9 Backwards Reduction and Emptiness Check

The algorithm uses a map from states to the set of rules that contain the state on their rhs.

```
  — Add an entry to the index
definition rqrm-add q r res ==
  case hm-lookup q res of
    None ⇒ hm-update q (ls-ins r (ls-empty ())) res |
    Some s ⇒ hm-update q (ls-ins r s) res


  — Lookup the set of rules with given state on rhs
definition rqrm-lookup rqrm q == case hm-lookup q rqrm of
  None ⇒ ls-empty () |
  Some s ⇒ s


  — Build the index from a set of rules
definition build-rqrm
  :: ('q::hashable,'l::hashable) ta-rule ls
     ⇒ ('q,('q,'l) ta-rule ls) hm
  where
  build-rqrm δ ==
    ls-iteratei δ (λ-. True)
      (λr res.
        foldl (λres q. rqrm-add q r res) res (rhsq r)
      )
      (hm-empty ())


— Whether the index satisfies the map and set invariants
definition rqrm-invar rqrm ==
  hm-invar rqrm ∧ (∀ q. ls-invar (rqrm-lookup rqrm q))
— Whether the index really maps a state to the set of rules with this state on their
```

```
    rhs
definition rqrm-prop δ rqrm ==
  ∀ q. ls-α (rqrm-lookup rqrm q) = {r∈δ. q∈set (rhsq r)}

lemma rqrm-α-lookup-update[simp]:
  rqrm-invar rqrm ⟹
    ls-α (rqrm-lookup (rqrm-add q r rqrm) q')
    = ( if q=q' then
          insert r (ls-α (rqrm-lookup rqrm q'))
        else
          ls-α (rqrm-lookup rqrm q')
      )
  by (simp
    add: rqrm-lookup-def rqrm-add-def rqrm-invar-def hm.correct
         ls.correct
    split: option.split-asm option.split)

lemma rqrm-propD:
  rqrm-prop δ rqrm ⟹ ls-α (rqrm-lookup rqrm q) = {r∈δ. q∈set (rhsq r)}
  by (simp add: rqrm-prop-def)

lemma build-rqrm-correct:
  fixes δ
  assumes [simp]: ls-invar δ
  shows rqrm-invar (build-rqrm δ) (is ?T1) and
        rqrm-prop (ls-α δ) (build-rqrm δ) (is ?T2)
proof −
  have rqrm-invar (build-rqrm δ) ∧
    (∀ q. ls-α (rqrm-lookup (build-rqrm δ) q) = {r∈ls-α δ. q∈set (rhsq r)})
    apply (unfold build-rqrm-def)
    apply (rule-tac
      I=λit res. (rqrm-invar res)
                 ∧ (∀ q. ls-α (rqrm-lookup res q)
                    = {r∈ls-α δ − it. q∈set (rhsq r)})
      in ls.iterate-rule-P)
      — Invar
    apply simp
      — Initial
    apply (simp add: hm-correct ls-correct rqrm-lookup-def rqrm-invar-def)
      — Step
    apply (rule-tac
      I=λres itl itr.
        (rqrm-invar res)
        ∧ (∀ q. ls-α (rqrm-lookup res q)
           = {r∈ls-α δ − it. q∈set (rhsq r)}
             ∪ {r. r=x ∧ q∈set itl})
      in Misc.foldl-rule-P)
        — Initial
      apply simp
```

```
    — Step
    apply (intro conjI)
    apply (simp
      add: rqrm-invar-def rqrm-add-def rqrm-lookup-def hm-correct
           ls-correct
      split: option.split option.split-asm)
    apply simp
    apply (simp
      add: rqrm-add-def rqrm-lookup-def hm-correct ls-correct
      split: option.split option.split-asm)
    apply (auto) [1]
    — Final
    apply auto [1]
    — Final
  apply simp
  done
thus ?T1 ?T2 by (simp-all add: rqrm-prop-def)
qed

— A state of the basic algorithm contains a set of discovered states, a worklist and
    a map from rules to the number of distinct states on its RHS that have not
    yet been discovered or are still on the worklist
type-synonym ('Q,'L) brc-state
= 'Q hs × 'Q list × (('Q,'L) ta-rule, nat) hm

— Abstraction to α'-level:
definition brc-α
:: ('Q::hashable,'L::hashable) brc-state ⇒ ('Q,'L) br'-state
where brc-α == λ(Q,W,rcm). (hs-α Q, set W, hm-α rcm)

definition brc-invar-add :: ('Q::hashable,'L::hashable) brc-state set
where
brc-invar-add == {(Q,W,rcm).
  hs-invar Q ∧
  distinct W ∧
  hm-invar rcm
  ∧ set W ⊆ hs-α Q}


definition brc-invar δ == brc-invar-add ∩ {s. brc-α s ∈ br'-invar δ}

definition brc-cond :: ('q::hashable,'l::hashable) brc-state ⇒ bool
where brc-cond == λ(Q,W,rcm). W≠[]

definition brc-inner-step
:: ('q,'l) ta-rule ⇒ ('q::hashable,'l::hashable) brc-state
    ⇒ ('q,'l) brc-state
where
brc-inner-step r == λ(Q,W,rcm).
```

```
    let c=the (hm-lookup r rcm);
        rcm' = hm-update r (c−(1::nat)) rcm;
        Q' = (if c ≤ 1 then hs-ins (lhs r) Q else Q);
        W' = (if c ≤ 1 ∧ ¬ hs-memb (lhs r) Q then lhs r # W else W) in
      (Q',W',rcm')

definition brc-step
  :: ('q,('q,'l) ta-rule ls) hm
      ⇒ ('q::hashable,'l::hashable) brc-state
      ⇒ ('q,'l) brc-state
where
  brc-step rqrm == λ(Q,W,rcm).
    ls-iteratei (rqrm-lookup rqrm (hd W)) (λ-. True) brc-inner-step
      (Q,tl W, rcm)

  — Initial concrete state
definition brc-iq :: ('q,'l) ta-rule ls ⇒ 'q::hashable hs
  where brc-iq δ == lh-set-xy.g-image-filter (λr.
    if rhsq r = [] then Some (lhs r) else None) δ

definition brc-rcm-init
  :: ('q::hashable,'l::hashable) ta-rule ls
      ⇒ (('q,'l) ta-rule,nat) hm
  where brc-rcm-init δ ==
    ls-iteratei δ (λ-. True)
      (λr res. hm-update r ((length (remdups (rhsq r)))) res)
      (hm-empty ())

definition brc-initial
  :: ('q::hashable,'l::hashable) ta-rule ls ⇒ ('q,'l) brc-state
  where brc-initial δ ==
    let iq=brc-iq δ in
      (iq, hs-to-list (iq), brc-rcm-init δ)

definition brc-det-algo rqrm δ == (|
  dwa-cond = brc-cond,
  dwa-step = brc-step rqrm,
  dwa-initial = brc-initial δ,
  dwa-invar = brc-invar (ls-α δ)
|)

  — Additional facts needed from the abstract level
lemma brc-inv-imp-WssQ: brc-α (Q,W,rcm)∈br'-invar δ ⟹ set W ⊆ hs-α Q
  by (auto simp add: brc-α-def br'-invar-def br'-α-def br-invar-def)

lemma brc-iq-correct:
  assumes [simp]: ls-invar δ
  shows hs-invar (brc-iq δ)
        hs-α (brc-iq δ) = br-iq (ls-α δ)
```

```
  by (auto simp add: brc-iq-def br-iq-def lh-set-xy.image-filter-correct)

lemma brc-rcm-init-correct:
  assumes INV[simp]: ls-invar δ
  shows r∈ls-α δ
    ⟹ hm-α (brc-rcm-init δ) r = Some ((card (set (rhsq r))))
  (is - ⟹ ?T1 r) and
    hm-invar (brc-rcm-init δ) (is ?T2)
proof −
  have G: (∀ r∈ls-α δ. ?T1 r) ∧ ?T2
    apply (unfold brc-rcm-init-def)
    apply (rule-tac
      I=λit res. hm-invar res
          ∧ (∀ r∈ls-α δ − it. hm-α res r = Some ((card (set (rhsq r)))))
      in ls.iterate-rule-P)
      — Invar
    apply simp
      — Init
    apply (auto simp add: hm-correct) [1]
      — Step
    apply (rule conjI)
      apply (simp add: hm.update-correct)

      apply (simp only: hm-correct hs-correct INV)
      apply (rule ballI)
      apply (case-tac r=x)
      apply (auto
        simp add: length-remdups-card
        intro!: arg-cong[where f=card]) [1]
      apply simp
      — Final
    apply simp
    done
  from G show ?T2 by auto
  fix r
  assume r∈ls-α δ
  thus ?T1 r using G by auto
qed

lemma brc-inner-step-br'-desc:
  ⟦ (Q,W,rcm)∈brc-invar δ ⟧ ⟹ brc-α (brc-inner-step r (Q,W,rcm)) = (
    if the (hm-α rcm r) ≤ 1 then
      insert (lhs r) (hs-α Q)
    else hs-α Q,
    if the (hm-α rcm r) ≤ 1 ∧ (lhs r) ∉ hs-α Q then
      insert (lhs r) (set W)
    else (set W),
    ((hm-α rcm)(r ↦ the (hm-α rcm r) − 1))
  )
```

```
  by (simp
    add: brc-invar-def brc-invar-add-def brc-α-def brc-inner-step-def Let-def
        hs-correct hm-correct)

lemma brc-step-invar:
  assumes RQRM: rqrm-invar rqrm
  shows ⟦ Σ∈brc-invar-add; brc-α Σ∈br′-invar δ; brc-cond Σ ⟧
        ⟹ (brc-step rqrm Σ)∈brc-invar-add
  apply (cases Σ)
  apply (simp add: brc-step-def)
  apply (rule-tac I=λit (Q,W,rcm). (Q,W,rcm)∈brc-invar-add ∧ set W ⊆ hs-α
Q
                in ls.iterate-rule-P)
  apply (simp add: RQRM[unfolded rqrm-invar-def])
  apply (case-tac b)
  apply (simp add: brc-invar-add-def distinct-tl brc-cond-def)
  apply (auto simp add: brc-invar-add-def distinct-tl brc-cond-def
            dest!: brc-inv-imp-WssQ) [1]
  apply (case-tac σ)
  apply (auto simp add: brc-invar-add-def br-invar-def brc-inner-step-def
                      Let-def hs-correct hm-correct) [1]
  apply (case-tac σ)
  apply simp
  done


lemma brc-step-abs:
  assumes RQRM: rqrm-invar rqrm    rqrm-prop δ rqrm
  assumes A: Σ∈brc-invar δ    brc-cond Σ
  shows (brc-α Σ, brc-α (brc-step rqrm Σ)) ∈ br′-step δ
proof −
  obtain Q W rcm where [simp]: Σ=(Q,W,rcm) by (cases Σ) auto
  from A show ?thesis
    apply (simp add: brc-step-def)
    apply (rule
      br′-inner-step-proof[OF ls.v1-iteratei-impl,
        where cinvar=λit (Q,W,rcm). (Q,W,rcm)∈brc-invar-add
                                  ∧ set W ⊆ hs-α Q and
            q=hd W])
    apply (case-tac W)
    apply (auto simp add: brc-cond-def brc-invar-add-def brc-invar-def
              dest!: brc-inv-imp-WssQ) [2]
    prefer 6
    apply (simp add: brc-α-def)
    apply (case-tac Σ)
    apply (auto
      simp add: brc-invar-def brc-invar-add-def brc-inner-step-def
              Let-def hm-correct hs-correct) [1]
    apply (auto
```

```
    simp add: brc-invar-add-def brc-inner-step-def brc-α-def
          br′-inner-step-def Let-def hm-correct hs-correct) [1]
  apply (simp add: RQRM[unfolded rqrm-invar-def])
  apply (simp add: rqrm-propD[OF RQRM(2)])
  apply (case-tac W)
  apply (simp-all add: brc-α-def brc-cond-def brc-invar-def) [2]
  apply (case-tac W)
  apply (simp-all add: brc-α-def brc-cond-def brc-invar-def
                brc-invar-add-def) [2]
  done
qed

lemma brc-initial-invar: ls-invar δ ⟹ (brc-initial δ)∈brc-invar-add
  by (simp
    add: brc-invar-add-def brc-initial-def brc-iq-correct Let-def
      brc-rcm-init-correct hs-correct)

lemma brc-cond-abs: brc-cond Σ ⟷ (brc-α Σ)∈br′-cond
  apply (cases Σ)
  apply (simp add: brc-cond-def br′-cond-def brc-α-def)
  done

lemma brc-initial-abs:
  ls-invar δ ⟹ brc-α (brc-initial δ) ∈ br′-initial (ls-α δ)
  by (auto
    simp add: brc-initial-def Let-def brc-α-def brc-iq-correct
          brc-rcm-init-correct hs-correct
    intro: br′-initial.intros)

lemma brc-pref-br′:
  assumes RQRM[simp]: rqrm-invar rqrm    rqrm-prop (ls-α δ) rqrm
  assumes INV[simp]: ls-invar δ
  shows wa-precise-refine (det-wa-wa (brc-det-algo rqrm δ))
                          (br′-algo (ls-α δ))
                          brc-α
  apply (unfold-locales)
  apply (simp-all add: brc-det-algo-def br′-algo-def det-wa-wa-def)
  apply (simp add: brc-cond-abs)
  apply (auto simp add: brc-step-abs[OF RQRM]) [1]
  apply (simp add: brc-initial-abs)
  apply (auto simp add: brc-invar-def) [1]
  apply (simp add: brc-cond-abs)
  done

lemma brc-while-algo:
  assumes RQRM[simp]: rqrm-invar rqrm    rqrm-prop (ls-α δ) rqrm
  assumes INV[simp]: ls-invar δ
  shows while-algo (det-wa-wa (brc-det-algo rqrm δ))
proof −
```

```
  from brc-pref-br'[OF RQRM INV] interpret
    ref: wa-precise-refine (det-wa-wa (brc-det-algo rqrm δ))
                           (br'-algo (ls-α δ))
                           brc-α .
  show ?thesis
    apply (rule ref.wa-intro)
    apply (simp add: br'-while-algo)
    apply (simp-all add: det-wa-wa-def brc-det-algo-def br'-algo-def)
    apply (simp add: brc-invar-def)
    apply (auto simp add: brc-step-invar) [1]
    apply (simp add: brc-initial-invar)
    done
qed

lemmas brc-det-while-algo =
  det-while-algo-intro[OF brc-while-algo]


lemma fst-brc-α: fst (brc-α s) = hs-α (fst s)
  by (cases s) (simp add: brc-α-def)

lemmas brc-invar-final =
  wa-precise-refine.transfer-correctness[OF
    brc-pref-br' br'-invar-final, unfolded fst-brc-α]

definition hta-bwd-reduce H ==
  let rqrm = build-rqrm (hta-δ H) in
    hta-reduce
      H
      (fst (while brc-cond (brc-step rqrm) (brc-initial (hta-δ H))))


theorem (in hashedTa) hta-bwd-reduce-correct:
  shows hta-α (hta-bwd-reduce H)
        = ta-reduce (hta-α H) (b-accessible (ls-α (hta-δ H))) (is ?T1)
        hashedTa (hta-bwd-reduce H) (is ?T2)
proof −
  interpret det-while-algo (brc-det-algo (build-rqrm δ) δ)
    by (rule brc-det-while-algo)
       (simp-all add: build-rqrm-correct)

  have LC: (while brc-cond (brc-step (build-rqrm δ)) (brc-initial δ)) = loop
    by (unfold loop-def)
       (simp add: brc-det-algo-def)

  from while-proof'[OF brc-invar-final] have
    G1: hs-α (fst loop) = b-accessible (ls-α δ)
    by (simp add: build-rqrm-correct)
  have G2: loop ∈ brc-invar (ls-α δ)
```

by (*rule while-proof*)
(*simp add*: *brc-det-algo-def*)
hence [*simp*]: *hs-invar* (*fst loop*)
by (*cases loop*)
(*simp add*: *brc-invar-def brc-invar-add-def*)

show *?T1 ?T2*
by (*simp-all add*: *hta-bwd-reduce-def LC hta-reduce-correct G1*)

qed

5.9.1 Emptiness Check with Witness Computation

definition *brec-construct-witness*
:: ('*q*::*hashable*,'*l*::*hashable tree*) *hm* ⇒ ('*q*,'*l*) *ta-rule* ⇒ '*l tree*
where *brec-construct-witness Qm r* ==
NODE (*rhsl r*) (*List.map* (λ*q*. *the* (*hm-lookup q Qm*)) (*rhsq r*))

lemma *brec-construct-witness-correct*:
⟦*hm-invar Qm*⟧ ⟹
brec-construct-witness Qm r = *construct-witness* (*hm-α Qm*) *r*
by (*auto*
simp add: *construct-witness-def brec-construct-witness-def hm-correct*)

type-synonym ('*Q*,'*L*) *brec-state*
= (('*Q*,'*L tree*) *hm*
× '*Q fifo*
× (('*Q*,'*L*) *ta-rule*, *nat*) *hm*
× '*Q option*)

— Abstractions
definition *brec-α*
:: ('*Q*::*hashable*,'*L*::*hashable*) *brec-state* ⇒ ('*Q*,'*L*) *brw-state*
where *brec-α* == λ(*Q*,*W*,*rcm*,*f*). (*hm-α Q*, *set* (*fifo-α W*), (*hm-α rcm*))

definition *brec-inner-step*
:: '*q hs* ⇒ ('*q*,'*l*) *ta-rule*
⇒ ('*q*::*hashable*,'*l*::*hashable*) *brec-state*
⇒ ('*q*,'*l*) *brec-state*
where *brec-inner-step Qi r* == λ(*Q*,*W*,*rcm*,*qwit*).
let c=*the* (*hm-lookup r rcm*);
cond = *c* ≤ *1* ∧ *hm-lookup* (*lhs r*) *Q* = *None*;
rcm' = *hm-update r* (*c*−(*1*::*nat*)) *rcm*;
Q' = (*if cond then*
hm-update (*lhs r*) (*brec-construct-witness Q r*) *Q*
else Q);
W' = (*if cond then fifo-enqueue* (*lhs r*) *W else W*);
qwit' = (*if c* ≤ *1* ∧ *hs-memb* (*lhs r*) *Qi then Some* (*lhs r*) *else qwit*)

```
    in
      (Q',W',rcm',qwit')

definition brec-step
  :: ('q,('q,'l) ta-rule ls) hm ⇒ 'q hs
      ⇒ ('q::hashable,'l::hashable) brec-state
      ⇒ ('q,'l) brec-state
  where brec-step rqrm Qi == λ(Q,W,rcm,qwit).
    let (q,W')=fifo-dequeue W in
      ls-iteratei (rqrm-lookup rqrm q) (λ-. True)
        (brec-inner-step Qi) (Q,W',rcm,qwit)


definition brec-iqm
  :: ('q::hashable,'l::hashable) ta-rule ls ⇒ ('q,'l tree) hm
  where brec-iqm δ ==
    ls-iteratei δ (λ-. True) (λr m. if rhsq r = [] then
                        hm-update (lhs r) (NODE (rhsl r) []) m
                      else m)
                (hm-empty ())

definition brec-initial
  :: 'q hs ⇒ ('q::hashable,'l::hashable) ta-rule ls
      ⇒ ('q,'l) brec-state
  where brec-initial Qi δ ==
  let iq=brc-iq δ in
    ( brec-iqm δ,
      hs-to-fifo.g-set-to-listr iq,
      brc-rcm-init δ,
      hh-set-xx.g-disjoint-witness iq Qi)

definition brec-cond
  :: ('q,'l) brec-state ⇒ bool
  where brec-cond == λ(Q,W,rcm,qwit). ¬ fifo-isEmpty W ∧ qwit = None

definition brec-invar-add
  :: 'Q set ⇒ ('Q::hashable,'L::hashable) brec-state set
  where
  brec-invar-add Qi == {(Q,W,rcm,qwit).
    hm-invar Q ∧
    distinct (fifo-α W) ∧
    hm-invar rcm ∧
    ( case qwit of
        None ⇒ Qi ∩ dom (hm-α Q) = {} |
        Some q ⇒ q∈Qi ∩ dom (hm-α Q))}


definition brec-invar Qi δ == brec-invar-add Qi ∩ {s. brec-α s ∈ brw-invar δ}
```

definition *brec-invar-inner Qi* ==
  brec-invar-add Qi ∩ {(*Q*,*W*,-,-). *set* (*fifo-α W*) ⊆ *dom* (*hm-α Q*)}

lemma *brec-invar-cons*:
  Σ∈*brec-invar Qi δ* ⟹ Σ∈*brec-invar-inner Qi*
  apply (*cases* Σ)
  apply (*simp add*: *brec-invar-def brw-invar-def br′-invar-def br-invar-def*
        brec-α-def brw-α-def br′-α-def brec-invar-inner-def)
  done

lemma *brec-brw-invar-cons*:
  brec-α Σ ∈ *brw-invar Qi* ⟹ *set* (*fifo-α* (*fst* (*snd* Σ))) ⊆ *dom* (*hm-α* (*fst* Σ))
  apply (*cases* Σ)
  apply (*simp add*: *brec-invar-def brw-invar-def br′-invar-def br-invar-def*
        brec-α-def brw-α-def br′-α-def)
  done

definition *brec-det-algo rqrm Qi δ* == (‖
  dwa-cond=*brec-cond*,
  dwa-step=*brec-step rqrm Qi*,
  dwa-initial=*brec-initial Qi δ*,
  dwa-invar=*brec-invar* (*hs-α Qi*) (*ls-α δ*)
‖)

lemma *brec-iqm-correct′*:
  assumes *INV*[*simp*]: *ls-invar δ*
  shows
    dom (*hm-α* (*brec-iqm δ*)) = {*lhs r* | *r*. *r*∈*ls-α δ* ∧ *rhsq r* = []} (**is** *?T1*)
    witness-prop (*ls-α δ*) (*hm-α* (*brec-iqm δ*)) (**is** *?T2*)
    hm-invar (*brec-iqm δ*) (**is** *?T3*)
proof −
  have *?T1* ∧ *?T2* ∧ *?T3*
    apply (*unfold brec-iqm-def*)
    apply (*rule-tac*
      I=λ*it m*. *hm-invar m*
          ∧ *dom* (*hm-α m*) = {*lhs r* | *r*. *r*∈*ls-α δ* − *it* ∧ *rhsq r* = []}
          ∧ *witness-prop* (*ls-α δ*) (*hm-α m*)
      in *ls.iterate-rule-P*)
    apply *simp*
    apply (*auto simp add*: *hm-correct witness-prop-def*) [*1*]
    apply (*auto simp add*: *hm-correct witness-prop-def*) [*1*]
    apply (*case-tac x*)
    apply (*auto intro*: *accs.intros*) [*1*]
    apply *simp*
    done
  thus *?T1 ?T2 ?T3* **by** *auto*
qed

lemma *brec-iqm-correct*:

```
  assumes INV[simp]: ls-invar δ
  shows hm-α (brec-iqm δ) ∈ brw-iq (ls-α δ)
proof −
  have (∀ q t. hm-α (brec-iqm δ) q = Some t
        ⟶ (∃ r∈ls-α δ. rhsq r = [] ∧ q = lhs r ∧ t = NODE (rhsl r) []))
      ∧ (∀ r∈ls-α δ. rhsq r = [] ⟶ hm-α (brec-iqm δ) (lhs r) ≠ None)
    apply (unfold brec-iqm-def)
    apply (rule-tac I=λit m. (
      (hm-invar m) ∧
      (∀ q t. hm-α m q = Some t
        ⟶ (∃ r∈ls-α δ. rhsq r = [] ∧ q = lhs r ∧ t = NODE (rhsl r) [])) ∧
      (∀ r∈ls-α δ−it. rhsq r = [] ⟶ hm-α m (lhs r) ≠ None)
      )
      in ls.iterate-rule-P)
    apply simp
    apply (simp add: hm-correct)
    apply (auto simp add: hm-correct) [1]
    apply (auto simp add: hm-correct) [1]
    done
  thus ?thesis by (blast intro: brw-iq.intros)
qed

lemma brec-inner-step-brw-desc:
  ⟦ Σ∈brec-invar-inner (hs-α Qi) ⟧
    ⟹ (brec-α Σ, brec-α (brec-inner-step Qi r Σ)) ∈ brw-inner-step r
  apply (cases Σ)
  apply (rule brw-inner-step.intros)
  apply (simp only: )
  apply (simp only: brec-α-def split-conv)
  apply (simp only: brec-inner-step-def brec-α-def Let-def split-conv)
  apply (auto
    simp add: brec-invar-inner-def brec-invar-add-def brec-α-def
              brec-inner-step-def
              Let-def hs-correct hm-correct fifo-correct
              brec-construct-witness-correct)
  done


lemma brec-step-invar:
  assumes RQRM: rqrm-invar rqrm    rqrm-prop δ rqrm
  assumes [simp]: hs-invar Qi
  shows ⟦ Σ∈brec-invar-add (hs-α Qi); brec-α Σ ∈ brw-invar δ;  brec-cond Σ ⟧
        ⟹ (brec-step rqrm Qi Σ)∈brec-invar-add (hs-α Qi)
  apply (frule brec-brw-invar-cons)
  apply (cases Σ)
  apply (simp add: brec-step-def fifo-correct)
  apply (case-tac fifo-α b)
  apply (simp
    add: brec-invar-def distinct-tl brec-cond-def fifo-correct
```

```
      )
  apply (rule-tac s=b in fifo.removelE)
  apply simp
  apply simp
  apply simp

  apply (rule-tac
    I=λit (Q,W,rcm,qwit). (Q,W,rcm,qwit)∈brec-invar-add (hs-α Qi)
                        ∧ set (fifo-α W) ⊆ dom (hm-α Q)
    in ls.iterate-rule-P)
  apply simp
  apply (simp
    add: brec-invar-def distinct-tl brec-cond-def fifo-correct
      )
  apply (simp
    add: brec-invar-def brec-invar-add-def distinct-tl brec-cond-def
      fifo-correct)
  apply (case-tac σ)
  apply (auto
    simp add: brec-invar-add-def brec-inner-step-def Let-def hs-correct
          hm-correct fifo-correct split: option.split-asm) [1]
  apply (case-tac σ)
  apply simp
  done

lemma brec-step-abs:
  assumes RQRM: rqrm-invar rqrm    rqrm-prop δ rqrm
  assumes INV[simp]: hs-invar Qi
  assumes A': Σ∈brec-invar (hs-α Qi) δ
  assumes COND: brec-cond Σ
  shows (brec-α Σ, brec-α (brec-step rqrm Qi Σ)) ∈ brw-step δ
proof −
  from A' have A: (brec-α Σ)∈brw-invar δ    Σ∈brec-invar-add (hs-α Qi)
    by (simp-all add: brec-invar-def)

  obtain Q W rcm qwit where [simp]: Σ=(Q,W,rcm,qwit) by (cases Σ) blast
  from A COND show ?thesis
    apply (simp add: brec-step-def fifo-correct)
    apply (case-tac fifo-α W)
    apply (simp
      add: brec-invar-def distinct-tl brec-cond-def fifo-correct
    )
    apply (rule-tac s=W in fifo.removelE)
    apply simp
    apply simp
    apply simp

    apply (rule brw-inner-step-proof[
      OF ls.v1-iteratei-impl,
```

```
      where cinvar=λit Σ. Σ∈brec-invar-inner (hs-α Qi) and
            q=hd (fifo-α W)])
    apply assumption
    apply (frule brec-brw-invar-cons)
    apply (simp-all
      add: brec-cond-def brec-invar-add-def fifo-correct
           brec-invar-inner-def) [1]
    prefer 6
    apply (simp add: brec-α-def)
    apply (case-tac Σ)
    apply (auto
      simp add: brec-invar-add-def brec-inner-step-def Let-def hm-correct
                hs-correct fifo-correct brec-invar-inner-def
      split: option.split-asm) [1]
    apply (blast intro: brec-inner-step-brw-desc)
    apply (simp add: RQRM[unfolded rqrm-invar-def])
    apply (simp
      add: rqrm-propD[OF RQRM(2)] fifo-correct)
    apply (simp-all
      add: brec-α-def brec-cond-def brec-invar-def fifo-correct) [1]
    apply (simp-all
      add: brec-α-def brec-cond-def brec-invar-add-def fifo-correct) [1]
    done
qed

lemma brec-invar-initial:
  ⟦ls-invar δ; hs-invar Qi⟧ ⟹ (brec-initial Qi δ) ∈ brec-invar-add (hs-α Qi)
  apply (auto
    simp add: brec-invar-add-def brec-initial-def brc-iq-correct
              brec-iqm-correct' hs-correct hs.isEmpty-correct Let-def
              brc-rcm-init-correct br-iq-def
              hh-set-xx.disjoint-witness-correct
              hs-to-fifo.g-set-to-listr-correct
    split: option.split)
  apply (auto simp add: brc-iq-correct
    hh-set-xx.disjoint-witness-None br-iq-def) [1]

  apply (drule hh-set-xx.disjoint-witness-correct[simplified])
  apply simp

  apply (drule hh-set-xx.disjoint-witness-correct[simplified])
  apply (simp add: brc-iq-correct br-iq-def)
  done

lemma brec-cond-abs:
  ⟦Σ∈brec-invar Qi δ⟧ ⟹ brec-cond Σ ⟷ (brec-α Σ)∈brw-cond Qi
  apply (cases Σ)
  apply (auto
    simp add: brec-cond-def brw-cond-def brec-α-def brec-invar-def
```

```
        brec-invar-add-def fifo-correct
    split: option.split-asm)
  done

lemma brec-initial-abs:
  ⟦ ls-invar δ; hs-invar Qi ⟧
    ⟹ brec-α (brec-initial Qi δ) ∈ brw-initial (ls-α δ)
  by (auto simp add: brec-initial-def Let-def brec-α-def
                     brc-iq-correct brc-rcm-init-correct brec-iqm-correct
                     br-iq-def fifo-correct hs-to-fifo.g-set-to-listr-correct
              intro: brw-initial.intros[unfolded br-iq-def])

lemma brec-pref-brw:
  assumes RQRM[simp]: rqrm-invar rqrm    rqrm-prop (ls-α δ) rqrm
  assumes INV[simp]: ls-invar δ    hs-invar Qi
  shows wa-precise-refine (det-wa-wa (brec-det-algo rqrm Qi δ))
                          (brw-algo (hs-α Qi) (ls-α δ))
                          brec-α
  apply (unfold-locales)
  apply (simp-all add: det-wa-wa-def brec-det-algo-def brw-algo-def)
  apply (auto simp add: brec-cond-abs brec-step-abs brec-initial-abs)
  apply (simp add: brec-invar-def)
  done

lemma brec-while-algo:
  assumes RQRM[simp]: rqrm-invar rqrm    rqrm-prop (ls-α δ) rqrm
  assumes INV[simp]: ls-invar δ    hs-invar Qi
  shows while-algo (det-wa-wa (brec-det-algo rqrm Qi δ))
proof −
  interpret ref:
    wa-precise-refine (det-wa-wa (brec-det-algo rqrm Qi δ))
                      (brw-algo (hs-α Qi) (ls-α δ))
                      brec-α
    using brec-pref-brw[OF RQRM INV] .

  show ?thesis
    apply (rule ref.wa-intro)
    apply (simp add: brw-while-algo)
    apply (simp-all add: det-wa-wa-def brec-det-algo-def brw-algo-def)
    apply (simp add: brec-invar-def)
    apply (auto simp add: brec-step-invar[OF RQRM INV(2)]) [1]
    apply (simp add: brec-invar-initial) [1]
    done
qed

lemma fst-brec-α: fst (brec-α Σ) = hm-α (fst Σ)
  by (cases Σ) (simp add: brec-α-def)

lemmas brec-invar-final =
```

```
  wa-precise-refine.transfer-correctness[
    OF brec-pref-brw brw-invar-final,
    unfolded fst-brec-α]

lemmas brec-det-algo = det-while-algo-intro[OF brec-while-algo]

definition hta-is-empty-witness H ==
  let rqrm = build-rqrm (hta-δ H);
      (Q,-,-,qwit) = (while brec-cond (brec-step rqrm (hta-Qi H))
                        (brec-initial (hta-Qi H) (hta-δ H)))
  in
    case qwit of
      None ⇒ None |
      Some q ⇒ (hm-lookup q Q)


theorem (in hashedTa) hta-is-empty-witness-correct:
  shows [rule-format]: hta-is-empty-witness H = Some t
                    ⟶ t∈ta-lang (hta-α H) (is ?T1)
       hta-is-empty-witness H = None ⟶ ta-lang (hta-α H) = {} (is ?T2)
proof −

  interpret det-while-algo (brec-det-algo (build-rqrm δ) Qi δ)
    by (rule brec-det-algo)
       (simp-all add: build-rqrm-correct)

  have LC:
    (while brec-cond (brec-step (build-rqrm δ) Qi) (brec-initial Qi δ)) = loop
    by (unfold loop-def)
       (simp add: brec-det-algo-def)

  from while-proof'[OF brec-invar-final] have X:
    hs-α Qi ∩ dom (hm-α (fst loop)) = {}
     ⟷ (hs-α Qi ∩ b-accessible (ls-α δ) = {})
    witness-prop (ls-α δ) (hm-α (fst loop))
    by (simp-all add: build-rqrm-correct)

  obtain Q W rcm qwit where
    [simp]: loop = (Q,W,rcm,qwit)
    by (case-tac loop) blast

  from loop-invar have I: loop ∈ brec-invar (hs-α Qi) (ls-α δ)
    by (simp add: brec-det-algo-def)
  hence INVARS[simp]: hm-invar Q    hm-invar rcm
    by (simp-all add: brec-invar-def brec-invar-add-def)

  {
    assume C: hta-is-empty-witness H = Some t
    then obtain q where
```

```
    [simp]: qwit=Some q and
      LUQ: hm-lookup q Q = Some t
    by (unfold hta-is-empty-witness-def)
      (simp add: LC split: option.split-asm)
  from LUQ have QqF: hm-α Q q = Some t by (simp add: hm-correct)
  from I have QMEM: q∈hs-α Qi
    by (simp-all add: brec-invar-def brec-invar-add-def)
  moreover from witness-propD[OF X(2)] QqF have accs (ls-α δ) t q by simp
  ultimately have t∈ta-lang (hta-α H)
    by (auto simp add: ta-lang-def hta-α-def)
 } moreover {
  assume C: hta-is-empty-witness H = None
  hence DJ: hs-α Qi ∩ dom (hm-α Q) = {} using I
    by (auto simp add: hta-is-empty-witness-def LC brec-invar-def
                       brec-invar-add-def hm-correct
              split: option.split-asm)
  with X have hs-α Qi ∩ b-accessible (ls-α δ) = {}
    by (simp add: brec-α-def)
  with empty-if-no-b-accessible[of hta-α H] have ta-lang (hta-α H) = {}
    by (simp add: hta-α-def)
 } ultimately show ?T1 ?T2 by auto
qed
```

5.10 Interface for Natural Number States and Symbols

The library-interface is statically instantiated to use natural numbers as both, states and symbols.

This interface is easier to use from ML and OCaml, because there is no overhead with typeclass emulation.

```
type-synonym htai = (nat,nat) hashedTa

definition htai-mem :: - ⇒ htai ⇒ bool
  where htai-mem == hta-mem
definition htai-prod :: htai ⇒ htai ⇒ htai
  where htai-prod H1 H2 == hta-reindex (hta-prod H1 H2)
definition htai-prodWR :: htai ⇒ htai ⇒ htai
  where htai-prodWR H1 H2 == hta-reindex (hta-prodWR H1 H2)
definition htai-union :: htai ⇒ htai ⇒ htai
  where htai-union H1 H2 == hta-reindex (hta-union H1 H2)
definition htai-empty :: unit ⇒ htai
  where htai-empty == hta-empty
definition htai-add-qi :: - ⇒ htai ⇒ htai
  where htai-add-qi == hta-add-qi
definition htai-add-rule :: - ⇒ htai ⇒ htai
  where htai-add-rule == hta-add-rule
definition htai-bwd-reduce :: htai ⇒ htai
  where htai-bwd-reduce == hta-bwd-reduce
definition htai-is-empty-witness :: htai ⇒ -
```

where *htai-is-empty-witness* == *hta-is-empty-witness*
definition *htai-ensure-idx-f* :: *htai* ⇒ *htai*
where *htai-ensure-idx-f* == *hta-ensure-idx-f*
definition *htai-ensure-idx-s* :: *htai* ⇒ *htai*
where *htai-ensure-idx-s* == *hta-ensure-idx-s*
definition *htai-ensure-idx-sf* :: *htai* ⇒ *htai*
where *htai-ensure-idx-sf* == *hta-ensure-idx-sf*

definition *htaip-prod* :: *htai* ⇒ *htai* ⇒ (*nat* ∗ *nat*,*nat*) *hashedTa*
where *htaip-prod* == *hta-prod*
definition *htaip-prodWR* :: *htai* ⇒ *htai* ⇒ (*nat* ∗ *nat*,*nat*) *hashedTa*
where *htaip-prodWR* == *hta-prodWR*
definition *htaip-reindex* :: (*nat* ∗ *nat*,*nat*) *hashedTa* ⇒ *htai*
where *htaip-reindex* == *hta-reindex*

locale *htai* = *hashedTa* +
constrains *H* :: *htai*
begin
lemmas *htai-mem-correct* = *hta-mem-correct*[*folded htai-mem-def*]

lemma *htai-empty-correct*[*simp*]:
hta-α (*htai-empty* ()) = *ta-empty*
hashedTa (*htai-empty* ())
by (*auto simp add*: *htai-empty-def hta-empty-correct*)

lemmas *htai-add-qi-correct* = *hta-add-qi-correct*[*folded htai-add-qi-def*]
lemmas *htai-add-rule-correct* = *hta-add-rule-correct*[*folded htai-add-rule-def*]

lemmas *htai-bwd-reduce-correct* =
hta-bwd-reduce-correct[*folded htai-bwd-reduce-def*]
lemmas *htai-is-empty-witness-correct* =
hta-is-empty-witness-correct[*folded htai-is-empty-witness-def*]

lemmas *htai-ensure-idx-f-correct* =
hta-ensure-idx-f-correct[*folded htai-ensure-idx-f-def*]
lemmas *htai-ensure-idx-s-correct* =
hta-ensure-idx-s-correct[*folded htai-ensure-idx-s-def*]
lemmas *htai-ensure-idx-sf-correct* =
hta-ensure-idx-sf-correct[*folded htai-ensure-idx-sf-def*]

end

lemma *htai-prod-correct*:
assumes [*simp*]: *hashedTa H1* *hashedTa H2*
shows
ta-lang (*hta-α* (*htai-prod H1 H2*)) = *ta-lang* (*hta-α H1*) ∩ *ta-lang* (*hta-α H2*)
hashedTa (*htai-prod H1 H2*)
apply (*unfold htai-prod-def*)
apply (*auto simp add*: *hta-prod-correct hashedTa.hta-reindex-correct*)

done

lemma *htai-prodWR-correct*:
assumes [*simp*]: *hashedTa H1* *hashedTa H2*
shows
ta-lang (*hta-α* (*htai-prodWR H1 H2*))
= *ta-lang* (*hta-α H1*) ∩ *ta-lang* (*hta-α H2*)
hashedTa (*htai-prodWR H1 H2*)
apply (*unfold htai-prodWR-def*)
apply (*auto simp add*: *hta-prodWR-correct hashedTa.hta-reindex-correct*)
done

lemma *htai-union-correct*:
assumes [*simp*]: *hashedTa H1* *hashedTa H2*
shows
ta-lang (*hta-α* (*htai-union H1 H2*))
= *ta-lang* (*hta-α H1*) ∪ *ta-lang* (*hta-α H2*)
hashedTa (*htai-union H1 H2*)
apply (*unfold htai-union-def*)
apply (*auto simp add*: *hta-union-correct hashedTa.hta-reindex-correct*)
done

5.11 Interface Documentation

This section contains a documentation of the executable tree-automata interface. The documentation contains a description of each function along with the relevant correctness lemmas.

ML/OCaml users should note, that there is an interface that has the fixed type Int for both states and function symbols. This interface is simpler to use from ML/OCaml than the generic one, as it requires no overhead to emulate Isabelle/HOL type-classes.

The functions of this interface start with the prefix *htai* instead of *hta*, but have the same semantics otherwise (cf Section 5.10).

5.11.1 Building a Tree Automaton

Function: *hta-empty*
Returns a tree automaton with no states and no rules.

Relevant Lemmas

hta-empty-correct: *hta-α* (*hta-empty* ()) = *ta-empty*

hashedTa (*hta-empty* ())

ta-empty-lang: *ta-lang ta-empty* = {}

Function: *hta-add-qi*
Adds an initial state to the given automaton.

Relevant Lemmas

hashedTa.hta-add-qi-correct $hashedTa\ H \Longrightarrow hta\text{-}\alpha\ (hta\text{-}add\text{-}qi\ qi\ H) = (\!|ta\text{-}initial = insert\ qi\ (ta\text{-}initial\ (hta\text{-}\alpha\ H)), ta\text{-}rules = ta\text{-}rules\ (hta\text{-}\alpha\ H)|\!)$

$hashedTa\ H \Longrightarrow hashedTa\ (hta\text{-}add\text{-}qi\ qi\ H)$

Function: *hta-add-rule*
Adds a rule to the given automaton.

Relevant Lemmas

hashedTa.hta-add-rule-correct: $hashedTa\ H \Longrightarrow hta\text{-}\alpha\ (hta\text{-}add\text{-}rule\ r\ H) = (\!|ta\text{-}initial = ta\text{-}initial\ (hta\text{-}\alpha\ H), ta\text{-}rules = insert\ r\ (ta\text{-}rules\ (hta\text{-}\alpha\ H))|\!)$

$hashedTa\ H \Longrightarrow hashedTa\ (hta\text{-}add\text{-}rule\ r\ H)$

5.11.2 Basic Operations

The tree automata of this library may have some optional indices, that accelerate computation. The tree-automata operations will compute the indices if necessary, but due to the pure nature of the Isabelle-language, the computed index cannot be stored for the next usage. Hence, before using a bulk of tree-automaton operations on the same tree-automata, the relevant indexes should be pre-computed.

Function: *hta-ensure-idx-f*
hta-ensure-idx-s
hta-ensure-idx-sf
Computes an index for a tree automaton, if it is not yet present.

Function: *hta-mem*, *hta-mem′*
Check whether a tree is accepted by the tree automaton.

Relevant Lemmas

hashedTa.hta-mem-correct: $hashedTa\ H \Longrightarrow hta\text{-}mem\ t\ H = (t \in ta\text{-}lang\ (hta\text{-}\alpha\ H))$

hashedTa.hta-mem′-correct: $[\![hashedTa\ H;\ hta\text{-}has\text{-}idx\text{-}f\ H]\!] \Longrightarrow hta\text{-}mem'\ t\ H = (t \in ta\text{-}lang\ (hta\text{-}\alpha\ H))$

Function: *hta-prod*, *hta-prod′*
Compute the product automaton. The computed automaton is in forward-reduced form. The language of the product automaton is the intersection of the languages of the two argument automata.

Relevant Lemmas

hta-prod-correct-aux: $[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hta\text{-}\alpha\ (hta\text{-}prod\ H1\ H2) = ta\text{-}fwd\text{-}reduce\ (ta\text{-}prod\ (hta\text{-}\alpha\ H1)\ (hta\text{-}\alpha\ H2))$

$[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prod\ H1\ H2)$

hta-prod-correct: $[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow ta\text{-}lang\ (hta\text{-}\alpha\ (hta\text{-}prod\ H1\ H2)) = ta\text{-}lang\ (hta\text{-}\alpha\ H1) \cap ta\text{-}lang\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prod\ H1\ H2)$

hta-prod′-correct-aux: $[\![hashedTa\ H1; hashedTa\ H2; hta\text{-}has\text{-}idx\text{-}s\ H1; hta\text{-}has\text{-}idx\text{-}sf\ H2]\!] \Longrightarrow hta\text{-}\alpha\ (hta\text{-}prod'\ H1\ H2) = ta\text{-}fwd\text{-}reduce\ (ta\text{-}prod\ (hta\text{-}\alpha\ H1)\ (hta\text{-}\alpha\ H2))$

$[\![hashedTa\ H1; hashedTa\ H2; hta\text{-}has\text{-}idx\text{-}s\ H1; hta\text{-}has\text{-}idx\text{-}sf\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prod'\ H1\ H2)$

hta-prod′-correct: $[\![hashedTa\ H1; hashedTa\ H2; hta\text{-}has\text{-}idx\text{-}s\ H1; hta\text{-}has\text{-}idx\text{-}sf\ H2]\!] \Longrightarrow ta\text{-}lang\ (hta\text{-}\alpha\ (hta\text{-}prod'\ H1\ H2)) = ta\text{-}lang\ (hta\text{-}\alpha\ H1) \cap ta\text{-}lang\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1; hashedTa\ H2; hta\text{-}has\text{-}idx\text{-}s\ H1; hta\text{-}has\text{-}idx\text{-}sf\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prod'\ H1\ H2)$

Function: *hta-prodWR*
Compute the product automaton by brute-force algorithm. The resulting automaton is not reduced. The language of the product automaton is the intersection of the languages of the two argument automata.

Relevant Lemmas

hta-prodWR-correct-aux: $[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hta\text{-}\alpha\ (hta\text{-}prodWR\ H1\ H2) = ta\text{-}prod\ (hta\text{-}\alpha\ H1)\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prodWR\ H1\ H2)$

hta-prodWR-correct: $[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow ta\text{-}lang\ (hta\text{-}\alpha\ (hta\text{-}prodWR\ H1\ H2)) = ta\text{-}lang\ (hta\text{-}\alpha\ H1) \cap ta\text{-}lang\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1; hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}prodWR\ H1\ H2)$

Function: *hta-union*
Compute the union of two tree automata.

Relevant Lemmas

hta-union-correct′: $[\![hashedTa\ H1;\ hashedTa\ H2]\!] \Longrightarrow hta\text{-}\alpha\ (hta\text{-}union\ H1\ H2) = ta\text{-}union\text{-}wrap\ (hta\text{-}\alpha\ H1)\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1;\ hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}union\ H1\ H2)$

hta-union-correct: $[\![hashedTa\ H1;\ hashedTa\ H2]\!] \Longrightarrow ta\text{-}lang\ (hta\text{-}\alpha\ (hta\text{-}union\ H1\ H2)) = ta\text{-}lang\ (hta\text{-}\alpha\ H1) \cup ta\text{-}lang\ (hta\text{-}\alpha\ H2)$

$[\![hashedTa\ H1;\ hashedTa\ H2]\!] \Longrightarrow hashedTa\ (hta\text{-}union\ H1\ H2)$

Function: *hta-reduce*
Reduce the automaton to the given set of states. All initial states outside this set will be removed. Moreover, all rules that contain states outside this set are removed, too.

Relevant Lemmas

hashedTa.hta-reduce-correct: $[\![hashedTa\ H;\ hs.invar\ Q]\!] \Longrightarrow hta\text{-}\alpha\ (hta\text{-}reduce\ H\ Q) = ta\text{-}reduce\ (hta\text{-}\alpha\ H)\ (hs.\alpha\ Q)$

$[\![hashedTa\ H;\ hs.invar\ Q]\!] \Longrightarrow hashedTa\ (hta\text{-}reduce\ H\ Q)$

Function: *hta-bwd-reduce*
Compute the backwards-reduced version of a tree automata. States from that no tree can be produced are removed. Backwards reduction does not change the language of the automaton.

Relevant Lemmas

hashedTa.hta-bwd-reduce-correct: $hashedTa\ H \Longrightarrow hta\text{-}\alpha\ (hta\text{-}bwd\text{-}reduce\ H) = ta\text{-}reduce\ (hta\text{-}\alpha\ H)\ (b\text{-}accessible\ (ls.\alpha\ (hta\text{-}\delta\ H)))$

$hashedTa\ H \Longrightarrow hashedTa\ (hta\text{-}bwd\text{-}reduce\ H)$

ta-reduce-b-acc: $ta\text{-}lang\ (ta\text{-}bwd\text{-}reduce\ TA) = ta\text{-}lang\ TA$

Function: *hta-is-empty-witness*
Check whether the language of the automaton is empty. If the language is not empty, a tree of the language is returned.
The following property is not (yet) formally proven, but should hold: If a tree is returned, the language contains no tree with a smaller depth than the returned one.

Relevant Lemmas

hashedTa.hta-is-empty-witness-correct: $[\![hashedTa\ H;\ hta\text{-}is\text{-}empty\text{-}witness\ H = Some\ t]\!] \Longrightarrow t \in ta\text{-}lang\ (hta\text{-}\alpha\ H)$

$[\![hashedTa\ H;\ hta\text{-}is\text{-}empty\text{-}witness\ H = None]\!] \Longrightarrow ta\text{-}lang\ (hta\text{-}\alpha\ H) = \{\}$

5.12 Code Generation

export-code
hta-mem hta-mem′ hta-prod hta-prod′ hta-prodWR hta-union
hta-empty hta-add-qi hta-add-rule
hta-reduce hta-bwd-reduce hta-is-empty-witness
hta-ensure-idx-f hta-ensure-idx-s hta-ensure-idx-sf

htai-mem htai-prod htai-prodWR htai-union
htai-empty htai-add-qi htai-add-rule
htai-bwd-reduce htai-is-empty-witness
htai-ensure-idx-f htai-ensure-idx-s htai-ensure-idx-sf

in *SML*
module-name *Ta*

export-code
hta-mem hta-mem′ hta-prod hta-prod′ hta-prodWR hta-union
hta-empty hta-add-qi hta-add-rule
hta-reduce hta-bwd-reduce hta-is-empty-witness
hta-ensure-idx-f hta-ensure-idx-s hta-ensure-idx-sf

htai-mem htai-prod htai-prodWR htai-union
htai-empty htai-add-qi htai-add-rule
htai-bwd-reduce htai-is-empty-witness
htai-ensure-idx-f htai-ensure-idx-s htai-ensure-idx-sf

in *Haskell*
module-name *Ta*
(*string-classes*)

export-code
hta-mem hta-mem′ hta-prod hta-prod′ hta-prodWR hta-union
hta-empty hta-add-qi hta-add-rule
hta-reduce hta-bwd-reduce hta-is-empty-witness
hta-ensure-idx-f hta-ensure-idx-s hta-ensure-idx-sf

htai-mem htai-prod htai-prodWR htai-union

htai-empty htai-add-qi htai-add-rule
htai-bwd-reduce htai-is-empty-witness
htai-ensure-idx-f htai-ensure-idx-s htai-ensure-idx-sf

in *OCaml*
module-name *Ta*

ML ‹
@{*code hta-mem*};
@{*code hta-mem'*};
@{*code hta-prod*};
@{*code hta-prod'*};
@{*code hta-prodWR*};
@{*code hta-union*};
@{*code hta-empty*};
@{*code hta-add-qi*};
@{*code hta-add-rule*};
@{*code hta-reduce*};
@{*code hta-bwd-reduce*};
@{*code hta-is-empty-witness*};
@{*code hta-ensure-idx-f*};
@{*code hta-ensure-idx-s*};
@{*code hta-ensure-idx-sf*};
@{*code htai-mem*};
@{*code htai-prod*};
@{*code htai-prodWR*};
@{*code htai-union*};
@{*code htai-empty*};
@{*code htai-add-qi*};
@{*code htai-add-rule*};
@{*code htai-bwd-reduce*};
@{*code htai-is-empty-witness*};
@{*code htai-ensure-idx-f*};
@{*code htai-ensure-idx-s*};
@{*code htai-ensure-idx-sf*};
(∗@{*code ls-size*};
@{*code hs-size*};
@{*code rs-size*}∗)
›

end

6 Conclusion

This development formalized basic tree automata algorithms and the class of tree-regular languages. Efficient code was generated for all the languages supported by the Isabelle2009 code generator, namely Standard-ML, OCaml, and Haskell.

6.1 Efficiency of Generated Code

The efficiency of the generated code, especially for Haskell, is quite good. On the author's dual-core machine with 2.6GHz and 4GiB memory, the generated code handles automata with several thousands rules and states in a few seconds. The Haskell-code is between 2 and 3 times slower than a Java-implementation of (approximately) the same algorithms.

A comparison to the Taml-library of the Timbuk-project [3] is not fair, because it runs in interpreted OCaml-Mode by default, and this is not comparable in speed to, e.g., compiled Haskell. However, the generated OCaml-code of our library can also be run in interpreted mode, to get a fair comparison with Taml:

The speed was compared for computing whether the intersection of two tree-automata is empty or not. The choice of this test was motivated by the author's requirements.

While our library also computes a witness for non-emptiness, the Taml-library has no such function. For some examples of non-empty languages, our library was about 14 times faster than Taml. This is mainly because our emptiness-test stops if the first initial state is found to be accessible, while the Timbuk-implementation always performs a complete reduction. However, even when compared for automata that have an empty language, i.e. where Timbuk and our library have to do the same work, our library was about 2 times faster.

There are some performance test cases with large, randomly created, automata in the directory *code*, that can be run by the script *doTests.sh*. These test cases read pairs of automata, intersect them and check the result for emptiness. If the intersection is not empty, a tree accepted by both automata is computed.

There are significant differences in efficiency between the used languages. Most notably, the Haskell code runs one order of magnitude faster than the SML and OCaml code. Also, using the more elaborated top-down intersection algorithm instead of the brute-forec algorithm brings the least performance gain in Haskell. The author suspects that the Haskell compiler does some optimization, perhaps by lazy-evaluation, that is missed by the ML systems.

6.2 Future Work

There are many starting points for improvement, some of which are mentioned below.

Implemented Algorithms In this development, only basic algorithms for non-deterministic tree-automata have been formalized. There are many more interesting algorithms and notions that may be formalized, amongst others tree transducers and minimization of (deterministic) tree automata.

Actually, the goal when starting this development was to implement, at least, intersection and emptiness check with witness computation. These algorithms are needed for a DPN[1] model checking algorithm[5] that the author is currently working on.

Refinement The algorithms are first formalized on an abstract level, and then manually refined to become executable. In theory, the abstract algorithms are already executable, as they involve only recursive functions and finite sets. We have experimented with simplifier setups to execute the algorithms in the simplifier, however the performance was quite bad and there where some problems with termination due to the innermost rewriting-strategy used by the simplifier, that required careful crafting of the simplifier setup.

The refinement is done in a somewhat systematic way, using the tools provided by the Isabelle Collections Framework (e.g. a data refinement framework for the while-combinator). However, most of the refinement work is done by hand, and the author believes that it should be possible to do the refinement with more tool support.

Another direction of future work would be to use the tree-automata framework developped here for applications. The author is currently working on a model-checker for DPNs that uses tree-automata based techniques [5], and plans to use this tree automata framework to generate a verified implementation of this model-checker. However, there are other interesting applications of tree automata, that could be formalized in Isabelle and, using this framework, be refined to efficient executable algorithms.

6.3 Trusted Code Base

In this section we shortly characterize on what our formal proof depends, i.e. how to interpret the information contained in this formal proof and the fact that it is accepted by the Isabelle/HOL system.

First of all, you have to trust the theorem prover and its axiomatization of HOL, the ML-platform, the operating system software and the hardware it

runs on. All these components are, in theory, able to cause false theorems to be proven. However, the probability of a false theorem to get proven due to a hardware error or an error in the operating system software is reasonably low. There are errors in hardware and operating systems, but they will usually cause the system to crash or exhibit other unexpected behaviour, instead of causing Isabelle to quitely accept a false theorem and behave normal otherwise. The theorem prover itself is a bit more critical in this aspect. However, Isabelle/HOL is implemented in LCF-style, i.e. all the proofs are eventually checked by a small kernel of trusted code, containing rather simple operations. HOL is the logic that is most frequently used with Isabelle, and it is unlikely that it's axiomatization in Isabelle is inconsistent and no one found and reported this inconsistency already.

The next crucial point is the code generator of Isabelle. We derive executable code from our specifications. The code generator contains another (thin) layer of untrusted code. This layer has some known deficiencies[2] (as of Isabelle2009) in the sense that invalid code is generated. This code is then rejected by the target language's compiler or interpreter, but does not silently compute the wrong thing.

Moreover, assuming correctness of the code generator, the generated code is only guaranteed to be *partially* correct[3], i.e. there are no formal termination guarantees.

Acknowledgements We thank Markus Müller-Olm for some interesting discussions. Moreover, we thank the people on the Isabelle mailing list for quickly giving useful answers to any Isabelle-related questions.

[2]For example, the Haskell code generator may generate variables starting with uppercase letters, while the Haskell-specification requires variables to start with lowercase letters. Moreover, the ML code generator does not know the ML value restriction, and may generate code that violates this restriction.

[3]A simple example is the always-diverging function $\mathsf{f_{div}} :: \mathsf{bool} = \mathsf{while}\ (\lambda x.\ \mathsf{True})\ \mathsf{id}\ \mathsf{True}$ that is definable in HOL. The lemma $\forall x.\ x = \mathsf{if}\ \mathsf{f_{div}}\ \mathsf{then}\ x\ \mathsf{else}\ x$ is provable in Isabelle and rewriting based on it could, theoretically, be inserted before the code generation process, resulting in code that always diverges

References

[1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.

[2] 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.

[3] T. Genet and V. V. T. Tong. Timbuk 2.2. Available on: http://www.grappa.univ-lille3.fr/tata.

[4] P. Lammich. Isabelle collection library. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/collections.shtml, 2009. Formal proof development.

[5] P. Lammich. Tree automata for analyzing dynamic pushdown networks. In J. Knoop and A. Prantl, editors, *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, number Bericht 2009-X-1. Technische Unversität Wien, 2009.