Tree Automata

Peter Lammich

March 17, 2025

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* → *l qs*) = *q*
fun *rhsq* **where** *rhsq* (*q* → *l qs*) = *qs*
fun *rhsl* **where** *rhsl* (*q* → *l qs*) = *l*
  — States in a rule
fun *rule-states* **where** *rule-states* (*q* → *l qs*) = *insert q* (*set qs*)
  — States in a set of rules
definition *δ-states δ* == ⋃(*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* $\delta\ t\ q$ 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 *Ta.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*
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma *rule-states-lhs*[*simp*]: *lhs r* ∈ *rule-states r*
  ⟨*proof*⟩

lemma *rule-states-rhsq*: *set* (*rhsq r*) ⊆ *rule-states r*
  ⟨*proof*⟩

lemma *rule-states-finite*[*simp*, *intro*!]: *finite* (*rule-states r*)
  ⟨*proof*⟩

lemma δ*-statesI*:
  assumes *A*: (*q* → *l qs*)∈δ
  shows *q*∈δ*-states* δ
    set qs ⊆ δ*-states* δ
  ⟨*proof*⟩

lemma δ*-statesI'*: ⟦(*q* → *l qs*)∈δ; *qi*∈*set qs*⟧ ⟹ *qi*∈δ*-states* δ
  ⟨*proof*⟩

lemma δ-*states-accsI*: *accs* δ *n* *q* $\Longrightarrow$ *q*$\in\delta$-*states* δ
⟨*proof*⟩

lemma δ-*states-union*[*simp*]: δ-*states* $(\delta\cup\delta')$ = δ-*states* δ $\cup$ δ-*states* δ'
⟨*proof*⟩

lemma δ-*states-insert*[*simp*]:
δ-*states* (*insert* *r* δ) = (*rule-states* *r* $\cup$ δ-*states* δ)
⟨*proof*⟩

lemma δ-*states-mono*: $[\![\delta \subseteq \delta']\!] \Longrightarrow$ δ-*states* δ $\subseteq$ δ-*states* δ'
⟨*proof*⟩

lemma δ-*states-finite*[*simp*, *intro*]: *finite* δ $\Longrightarrow$ *finite* (δ-*states* δ)
⟨*proof*⟩

lemma δ-*statesE*: $[\![$*q*$\in\delta$-*states* Δ;
!!*f* *qs*. $[\![$ (*q* $\rightarrow$ *f* *qs*)$\in\Delta$ $]\!] \Longrightarrow$ *P*;
!!*ql* *f* *qs*. $[\![$ (*ql* $\rightarrow$ *f* *qs*)$\in\Delta$; *q*$\in$*set* *qs* $]\!] \Longrightarrow$ *P*
$]\!] \Longrightarrow$ *P*
⟨*proof*⟩

lemma δ-*symbolsI*: (*q* $\rightarrow$ *f* *qs*)$\in\delta$ $\Longrightarrow$ *f*$\in\delta$-*symbols* δ
⟨*proof*⟩

lemma δ-*symbolsE*:
assumes *A*: *f*$\in\delta$-*symbols* δ
obtains *q* *qs* **where** (*q* $\rightarrow$ *f* *qs*) $\in$ δ
⟨*proof*⟩

lemma δ-*symbols-simps*[*simp*]:
δ-*symbols* {} = {}
δ-*symbols* (*insert* *r* δ) = *insert* (*rhsl* *r*) (δ-*symbols* δ)
δ-*symbols* $(\delta\cup\delta')$ = δ-*symbols* δ $\cup$ δ-*symbols* δ'
⟨*proof*⟩

lemma δ-*symbols-finite*[*simp*, *intro*!]:
finite δ $\Longrightarrow$ *finite* (δ-*symbols* δ)
⟨*proof*⟩

lemma *accs-mono*: $[\![$*accs* δ *n* *q*; $\delta\subseteq\delta']\!] \Longrightarrow$ *accs* δ' *n* *q*
⟨*proof*⟩

context *tree-automaton*
begin
lemma *initial-subset*: *ta-initial* *TA* $\subseteq$ *ta-rstates* *TA*
⟨*proof*⟩
lemma *states-subset*: δ-*states* (*ta-rules* *TA*) $\subseteq$ *ta-rstates* *TA*
⟨*proof*⟩

lemma *finite-states*[*simp*, *intro*!]: *finite* (*ta-rstates TA*)
⟨*proof*⟩

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

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

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

locale *finite-alphabet* =
fixes *A* :: ($'L \rightharpoonup nat$)
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*
⟨*proof*⟩

lemma *legal-rules-finite*[*simp*, *intro*!]:
fixes *Q*::$'Q$ *set*
assumes [*simp*, *intro*!]: *finite Q*
shows *finite* (*legal-rules Q*)
⟨*proof*⟩
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*: $r{\in}\delta \Longrightarrow r{\in}legal\text{-}rules\ Q$
⟨*proof*⟩
lemma *accs-is-ranked*: $accs\ \delta\ t\ q \Longrightarrow t{\in}ranked\text{-}trees\ A$
⟨*proof*⟩
theorem *lang-is-ranked*: $ta\text{-}lang\ TA \subseteq ranked\text{-}trees\ A$
⟨*proof*⟩

end

3.3.2 Deterministic Tree Automata

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*: $[\![\ accs\ \delta\ t\ q;\ accs\ \delta\ t\ q'\ ]\!] \Longrightarrow q{=}q'$
⟨*proof*⟩

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*:
$[\![\ qs{\in}lists\ Q;\ A\ f = Some\ (length\ qs)\ ]\!] \Longrightarrow \exists\, q.\ (q \to f\ qs){\in}\delta$
begin

— In a complete DFTA, all trees can be labeled by some state
theorem *label-all*: $t{\in}ranked\text{-}trees\ A \Longrightarrow \exists\, q{\in}Q.\ accs\ \delta\ t\ q$
⟨*proof*⟩

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* = {}
⟨*proof*⟩

theorem *ta-empty-ta*[*simp*, *intro*!]: *tree-automaton ta-empty*
⟨*proof*⟩

theorem (**in** *finite-alphabet*) *ta-empty-rta*[*simp*, *intro*!]:
ranked-tree-automaton ta-empty A
⟨*proof*⟩

theorem (**in** *finite-alphabet*) *ta-empty-dta*[*simp*, *intro*!]:
det-tree-automaton ta-empty A
⟨*proof*⟩

3.4.2 Remapping of States

fun *remap-rule* **where** *remap-rule f* ($q \to l\ qs$) = (($f\ q$) $\to l$ (*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
⟨*proof*⟩

lemma *remap-accs1*: *accs* δ *n q* $\Longrightarrow$ *accs* (*remap-rule f* ' δ) *n* (*f q*)
⟨*proof*⟩

lemma *remap-lang1*: *t*∈*ta-lang TA* $\Longrightarrow$ *t*∈*ta-lang* (*ta-remap f TA*)
⟨*proof*⟩

lemma *remap-accs2*: ⟦
accs δ' *n* q';
δ'=(*remap-rule f* ' δ);
q'=*f q*;
inj-on f Q;
q∈*Q*;
δ*-states* $\delta \subseteq Q$
⟧ $\Longrightarrow$ *accs* δ *n q*
⟨*proof*⟩

lemma (**in** *tree-automaton*) *remap-lang2*:
assumes *I*: *inj-on f* (*ta-rstates TA*)
shows *t*∈*ta-lang* (*ta-remap f TA*) $\Longrightarrow$ *t*∈*ta-lang TA*
⟨*proof*⟩

theorem (**in** *tree-automaton*) *remap-lang*:
$inj\text{-}on\ f\ (ta\text{-}rstates\ TA) \Longrightarrow ta\text{-}lang\ (ta\text{-}remap\ f\ TA) = ta\text{-}lang\ TA$
⟨*proof*⟩

lemma (**in** *tree-automaton*) *remap-ta*[*intro*!, *simp*]:
tree-automaton (*ta-remap f TA*)
⟨*proof*⟩

lemma (**in** *ranked-tree-automaton*) *remap-rta*[*intro*!, *simp*]:
ranked-tree-automaton (*ta-remap f TA*) *A*
⟨*proof*⟩

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*⟩

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*⟩

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*:
$[\![\ accs\ \delta n\ n\ q;\ \delta n{=}\delta{\cup}\delta';\ \delta\text{-}states\ \delta \cap \delta\text{-}states\ \delta' = \{\};\ q{\in}\delta\text{-}states\ \delta\ ]\!]$
$\Longrightarrow accs\ \delta\ n\ q$
⟨*proof*⟩

corollary *accs-exclusive1*:
$[\![\ accs\ (\delta{\cup}\delta')\ n\ q;\ \delta\text{-}states\ \delta \cap \delta\text{-}states\ \delta' = \{\};\ q{\in}\delta\text{-}states\ \delta\ ]\!]$
$\Longrightarrow accs\ \delta\ n\ q$
⟨*proof*⟩

corollary *accs-exclusive2*:
$[\![\ accs\ (\delta{\cup}\delta')\ n\ q;\ \delta\text{-}states\ \delta \cap \delta\text{-}states\ \delta' = \{\};\ q{\in}\delta\text{-}states\ \delta'\ ]\!]$
$\Longrightarrow accs\ \delta'\ n\ q$
⟨*proof*⟩

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*⟩

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*⟩
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′*)
⟨*proof*⟩

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*⟩

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* = {}
⟨*proof*⟩

lemma *states-usw-disjoint*[*simp*]:
ta-rstates (*ta-remap USW1 X*) ∩ *ta-rstates* (*ta-remap USW2 Y*) = {}
⟨*proof*⟩

lemma *usw-inj-on*[*simp*, *intro*!]:
inj-on USW1 X
inj-on USW2 X
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

lemma *reduce-rulesD*:
⟦ *r*∈*reduce-rules* δ *P* ⟧ ⟹ *r*∈δ
⟦ *r*∈*reduce-rules* δ *P*; *q*∈*rule-states r*⟧ ⟹ *q*∈*P*
⟨*proof*⟩

lemma *reduce-rules-subset*: *reduce-rules* δ *P* ⊆ δ
⟨*proof*⟩

lemma *reduce-rules-mono*: *P* ⊆ *P'* ⟹ *reduce-rules* δ *P* ⊆ *reduce-rules* δ *P'*
⟨*proof*⟩

lemma δ*-states-reduce-subset*:
shows δ*-states* (*reduce-rules* δ *Q*) ⊆ δ*-states* δ ∩ *Q*
⟨*proof*⟩

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*⟩

lemma *reduce-δ-states-rules*[*simp*]:
(*ta-rules* (*ta-reduce TA* (*δ-states* (*ta-rules TA*)))) = *ta-rules TA*
⟨*proof*⟩
lemma *ta-reduce-δ-states*:
ta-lang (*ta-reduce TA* (*δ-states* (*ta-rules TA*))) = *ta-lang TA*
⟨*proof*⟩

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 $q \leftarrow f(\ldots q' \ldots)$.

inductive-set *f-succ* **for** *δ* **where**
⟦(*q* → *l qs*)∈*δ*; *q*′∈*set qs*⟧ ⟹ (*q*,*q*′) ∈ *f-succ δ*

— Alternative characterization of forward successors
lemma *f-succ-alt*: *f-succ δ* = {(*q*,*q*′). ∃ *l qs*. (*q* → *l qs*)∈*δ* ∧ *q*′∈*set qs*}
⟨*proof*⟩
definition *f-accessible δ Q0* == ((*f-succ δ*)*) ‘‘ *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-accessible-alt* :: (′*Q*,′*L*) *ta-rule set* ⇒ ′*Q set* ⇒ ′*Q set*
for *δ Q0*
where
fa-refl: *q0*∈*Q0* ⟹ *q0* ∈ *f-accessible-alt δ Q0* |
fa-step: ⟦ *q*∈*f-accessible-alt δ Q0*; (*q* → *l qs*)∈*δ*; *q*′∈*set qs* ⟧
⟹ *q*′ ∈ *f-accessible-alt δ Q0*

lemma *f-accessible-alt*: *f-accessible δ Q0* = *f-accessible-alt δ Q0*
⟨*proof*⟩

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 δ* ⟹ *finite* (*f-succ δ*)
⟨*proof*⟩

lemma *f-accessible-mono*: *Q*⊆*Q*′ ⟹ *x*∈*f-accessible δ Q* ⟹ *x*∈*f-accessible δ Q*′
⟨*proof*⟩

lemma *f-accessible-prepend*:

⟦ $(q \to l\ qs) \in \delta$; $q' \in set\ qs$; $x \in f\text{-}accessible\ \delta\ \{q'\}$ ⟧
$\Longrightarrow x \in f\text{-}accessible\ \delta\ \{q\}$
⟨*proof*⟩

lemma *f-accessible-subset*: $q \in f\text{-}accessible\ \delta\ Q \Longrightarrow q \in Q \cup \delta\text{-}states\ \delta$
⟨*proof*⟩

lemma (**in** *tree-automaton*) *f-accessible-in-states*:
$q \in f\text{-}accessible\ (ta\text{-}rules\ TA)\ (ta\text{-}initial\ TA) \Longrightarrow q \in ta\text{-}rstates\ TA$
⟨*proof*⟩

lemma *f-accessible-refl-inter-simp*[*simp*]: $Q \cap f\text{-}accessible\ r\ Q = Q$
⟨*proof*⟩
lemma *accs-reduce-f-acc*:
$accs\ \delta\ t\ q \Longrightarrow accs\ (reduce\text{-}rules\ \delta\ (f\text{-}accessible\ \delta\ \{q\}))\ t\ q$
⟨*proof*⟩
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*
⟨*proof*⟩

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* :: $('Q,'L)\ ta\text{-}rule\ set \Rightarrow 'Q\ set$
for δ
where
⟦ $(q \to l\ qs) \in \delta$; $!!x.\ x \in set\ qs \Longrightarrow x \in b\text{-}accessible\ \delta$ ⟧ $\Longrightarrow q \in b\text{-}accessible\ \delta$

lemma *b-accessibleI*:
⟦$(q \to l\ qs) \in \delta$; $set\ qs \subseteq b\text{-}accessible\ \delta$⟧ $\Longrightarrow q \in b\text{-}accessible\ \delta$
⟨*proof*⟩
lemma *accs-is-b-accessible*: $accs\ \delta\ t\ q \Longrightarrow q \in b\text{-}accessible\ \delta$
⟨*proof*⟩

lemma *b-acc-subset-δ-statesI*: $x \in b\text{-}accessible\ \delta \Longrightarrow x \in \delta\text{-}states\ \delta$
⟨*proof*⟩

lemma *b-acc-subset-δ-states*: $b\text{-}accessible\ \delta \subseteq \delta\text{-}states\ \delta$
⟨*proof*⟩

lemma *b-acc-finite*[*simp*, *intro*!]: $finite\ \delta \Longrightarrow finite\ (b\text{-}accessible\ \delta)$
⟨*proof*⟩

lemma *b-accessible-is-accs*:
  ⟦ *q*∈*b-accessible* (*ta-rules TA*);
    !!*t*. *accs* (*ta-rules TA*) *t q* ⟹ *P*
  ⟧ ⟹ *P*
⟨*proof*⟩
lemma *accs-reduce-b-acc*:
  accs δ *t q* ⟹ *accs* (*reduce-rules* δ (*b-accessible* δ)) *t q*
  ⟨*proof*⟩
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*
  ⟨*proof*⟩
theorem *empty-if-no-b-accessible*:
  ta-lang TA = {} ⟷ *ta-initial TA* ∩ *b-accessible* (*ta-rules TA*) = {}
  ⟨*proof*⟩

3.4.5 Product Automaton

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

fun *r-prod* **where**
  r-prod (*q1* → *l1 qs1*) (*q2* → *l2 qs2*) = ((*q1*,*q2*) → *l1* (*zip qs1 qs2*))

  — Product rules
definition δ*-prod* δ*1* δ*2* == {
  r-prod (*q1* → *l qs1*) (*q2* → *l qs2*) | *q1 q2 l qs1 qs2*.
    length qs1 = *length qs2* ∧
    (*q1* → *l qs1*)∈δ*1* ∧
    (*q2* → *l qs2*)∈δ*2*
}

lemma δ*-prodI*: ⟦
    length qs1 = *length qs2*;
    (*q1* → *l qs1*)∈δ*1*;
    (*q2* → *l qs2*)∈δ*2* ⟧ ⟹ ((*q1*,*q2*) → *l* (*zip qs1 qs2*)) ∈ δ*-prod* δ*1* δ*2*
  ⟨*proof*⟩

lemma δ*-prodE*:
  ⟦
    r∈δ*-prod* δ*1* δ*2*;
    !!*q1 q2 l qs1 qs2*. ⟦ *length qs1* = *length qs2*;
                          (*q1* → *l qs1*)∈δ*1*;
                          (*q2* → *l qs2*)∈δ*2*;
                          r = ((*q1*,*q2*) → *l* (*zip qs1 qs2*))
                        ⟧ ⟹ *P*
  ⟧ ⟹ *P*
  ⟨*proof*⟩
lemma δ*-prod-sound*:

assumes *A*: *accs* (*δ-prod δ1 δ2*) *t* (*q1*,*q2*)
shows *accs δ1 t q1*   *accs δ2 t q2*
⟨*proof*⟩
lemma *δ-prod-precise*:
⟦ *accs δ1 t q1*; *accs δ2 t q2* ⟧ ⟹ *accs* (*δ-prod δ1 δ2*) *t* (*q1*,*q2*)
⟨*proof*⟩

lemma *δ-prod-empty*[*simp*]:
δ-prod {} *δ* = {}
δ-prod δ {} = {}
⟨*proof*⟩

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*}
⟨*proof*⟩

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′*
⟨*proof*⟩

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*}
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *δ-states-cart*:
  q ∈ *δ-states* (*δ-prod δ1 δ2*) ⟹ *q* ∈ *δ-states δ1* × *δ-states δ2*
  ⟨*proof*⟩

lemma *δ-prod-finite* [*simp*, *intro*]:
  finite δ1 ⟹ *finite δ2* ⟹ *finite* (*δ-prod δ1 δ2*)
⟨*proof*⟩

lemma *ta-prod-correct-aux2*:
  assumes *TA*: *tree-automaton TA1*   *tree-automaton TA2*
  shows *tree-automaton* (*ta-prod TA1 TA2*)
⟨*proof*⟩
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*)
  ⟨*proof*⟩

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*⟩

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 *δss* :: (*'Q set*,*'L*) *ta-rule set* **where**

$[\![$ *A f* = *Some* (*length ss*);
ss ∈ *lists* {*s*. *s* ⊆ *ta-rstates TA*};
s = *δss-lhs f ss*
$]\!] \Longrightarrow$ (*s* → *f ss*) ∈ *δss*

lemma *δssI*:
assumes *A*: *A f* = *Some* (*length ss*)
ss ∈ *lists* {*s*. *s* ⊆ *ta-rstates TA*}
shows
((*δss-lhs f ss*) → *f ss*) ∈ *δss*
⟨*proof*⟩

lemma *δss-subset*[*simp*, *intro*!]: *δss-lhs f ss* ⊆ *Q*
⟨*proof*⟩

lemma *δss-finite*[*simp*, *intro*!]: *finite δss*
⟨*proof*⟩

lemma *δss-det*: $[\![$ (*q* → *f qs*) ∈ *δss*; (*q*′ → *f qs*) ∈*δss* $]\!] \Longrightarrow$ *q*=*q*′
⟨*proof*⟩

lemma *δss-accs-sound*:
assumes *A*: *accs δ t q*
obtains *s* **where**
s⊆*Q*
q∈*s*
accs δss t s
⟨*proof*⟩

lemma *δss-accs-precise*:
assumes *A*: *accs δss t s* *q*∈*s*
shows *accs δ t q*
⟨*proof*⟩
definition *detTA* == (| *ta-initial* = { *s*. *s*⊆*Q* ∧ *s*∩*Qi* ≠ {} },
ta-rules = *δss* |)

theorem *detTA-is-ta*[*simp*, *intro*]:
det-tree-automaton detTA A
⟨*proof*⟩

theorem *detTA-lang*[*simp*]:
ta-lang (*detTA*) = *ta-lang TA*
⟨*proof*⟩

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*
⟨*proof*⟩
definition δ*complete* :: ('*Q option*, '*L*) *ta-rule set* **where**
δ*complete* == (*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* δ*complete* ⟹ *q*∈*Qcomplete*
⟨*proof*⟩

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

lemma δ*complete-finite*[*simp*, *intro*]: *finite* δ*complete*
⟨*proof*⟩

theorem *completeTA-is-ta*: *complete-tree-automaton completeTA A*
⟨*proof*⟩

theorem *completeTA-lang*: *ta-lang completeTA* = *ta-lang TA*
⟨*proof*⟩

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
⟨*proof*⟩

theorem *complementTA-correct*:
ta-lang complementTA = *ranked-trees A* − *ta-lang TA* (**is** *?T1*)
complete-tree-automaton complementTA A (**is** *?T2*)
⟨*proof*⟩

end

3.5 Regular Tree Languages

3.5.1 Definitions

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
⟨*proof*⟩

context *ranked-tree-automaton*
begin

lemma (**in** *ranked-tree-automaton*) *rtlI*[*simp*]:
shows *ta-lang TA* ∈ *regular-languages A*
⟨*proof*⟩

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*⟩

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*⟩

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*
  ⟨*proof*⟩

theorem *rtl-union-closed*:
  ⟦ *L1∈regular-languages A*; *L2∈regular-languages A* ⟧
    ⟹ *L1∪L2* ∈ *regular-languages A*
⟨*proof*⟩

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

theorem *rtl-complement-closed*:
  L∈regular-languages A ⟹ *ranked-trees A* − *L* ∈ *regular-languages A*
⟨*proof*⟩

theorem (**in** *finite-alphabet*) *rtl-univ*:
  ranked-trees A ∈ *regular-languages A*
  ⟨*proof*⟩

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*⟩

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** ‹→› *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* ⇒ *'a list* ⇒ *bool* **where**
  r-match [] [] ⟷ *True* |
  r-match (*A#AS*) (*a#as*) ⟷ *a*∈*A* ∧ *r-match AS as* |
  r-match - - ⟷ *False*

— *AbsAlgo.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 ⟷ *length L* = *length l* ∧ (∀ *i*<*length l*. *l*!*i* ∈ *L*!*i*)
  ⟨*proof*⟩
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*⟩

theorem *faccs-correct1*: $q \in faccs\ \delta\ n \Longrightarrow accs\ \delta\ n\ q$
⟨*proof*⟩
theorem *faccs-correct2*: $accs\ \delta\ n\ q \Longrightarrow q \in faccs\ \delta\ n$
⟨*proof*⟩

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

lemma *faccs-alt*: $faccs\ \delta\ t = \{q.\ accs\ \delta\ t\ q\}$ ⟨*proof*⟩

4.2 Backward Reduction and Emptiness Check

4.2.1 Auxiliary Definitions

inductive-set *bacc-step* :: $('Q,'L)$ *ta-rule set* ⇒ *'Q set* ⇒ *'Q set*
for $\delta\ Q$
where
$[\![\ r \in \delta;\ set\ (rhsq\ r) \subseteq Q\ ]\!] \Longrightarrow lhs\ r \in bacc\text{-}step\ \delta\ Q$

— 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*: $bacc\text{-}step\ \delta\ Q \subseteq Q$
shows $b\text{-}accessible\ \delta \subseteq Q$
⟨*proof*⟩

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: **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*
$:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow 'Q \Rightarrow ('Q,'L)\ br\text{-}state \Rightarrow 'Q\ set$
where
$br\text{-}dsq\ \delta\ q == \lambda(Q,W).\ \{\ lhs\ r\ |\ r.\ r{\in}\delta \wedge set\ (rhsq\ r) \subseteq (Q{-}(W{-}\{q\}))\ \}$

— 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*
$:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow (('Q,'L)\ br\text{-}state \times ('Q,'L)\ br\text{-}state)\ set$
for δ **where**
$[\![$
$q{\in}W;$
$Q' = Q \cup br\text{-}dsq\ \delta\ q\ (Q,W);$
$W' = W - \{q\} \cup (br\text{-}dsq\ \delta\ q\ (Q,W) - Q)$
$]\!] \Longrightarrow ((Q,W),(Q',W')){\in}br\text{-}step\ \delta$

— Termination condition for backwards reduction: The workset is empty

definition *br-cond* $:: ('Q,'L)\ br\text{-}state\ set$
where $br\text{-}cond == \{(Q,W).\ W{\neq}\{\}\}$

— Termination condition for emptiness check: The workset is empty or a non-empty initial state has been discovered

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

— Set of all states that occur on the lhs of a constant-rule

definition *br-iq* $:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow 'Q\ set$
where $br\text{-}iq\ \delta == \{\ lhs\ r\ |\ r.\ r{\in}\delta \wedge rhsq\ r = []\ \}$

— Initial state for the iteration

definition *br-initial* $:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow ('Q,'L)\ br\text{-}state$
where $br\text{-}initial\ \delta == (br\text{-}iq\ \delta,\ br\text{-}iq\ \delta)$

— 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* $:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow ('Q,'L)\ br\text{-}state\ set$
where $br\text{-}invar\ \delta == \{(Q,W).$
$W{\subseteq}Q\ \wedge$

$Q \subseteq$ *b-accessible* $\delta \wedge$
bacc-step δ $(Q - W) \subseteq Q \wedge$
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 ==
$(\{(Q',Q).\ Q \subset Q' \wedge Q' \subseteq$ *b-accessible* $\delta\})$ <**lex**> *finite-psubset*

lemma *bre-cond-imp-br-cond*[*intro*, *simp*]: *bre-cond Qi* $\subseteq$ *br-cond*
⟨*proof*⟩

lemma *br-termrel-wf*[*simp*, *intro*!]: *finite* $\delta \Longrightarrow$ *wf* (*br-termrel* δ)
⟨*proof*⟩
lemma *br-dsq-ss*:
assumes *A*: $(Q,W) \in$ *br-invar* δ $\quad W \neq \{\}$ $\quad q \in W$
shows *br-dsq* δ q $(Q,W) \subseteq$ *b-accessible* δ
⟨*proof*⟩

lemma *br-step-in-termrel*:
assumes *A*: $\Sigma \in$ *br-cond* $\quad \Sigma \in$ *br-invar* δ $\quad (\Sigma,\Sigma') \in$ *br-step* δ
shows $(\Sigma', \Sigma) \in$ *br-termrel* δ
⟨*proof*⟩

lemma *br-invar-initial*[*simp*]: *finite* $\delta \Longrightarrow$ (*br-initial* δ)$\in$*br-invar* δ
⟨*proof*⟩

lemma *br-invar-step*:
assumes [*simp*]: *finite* δ
assumes *A*: $\Sigma \in$ *br-cond* $\quad \Sigma \in$ *br-invar* δ $\quad (\Sigma,\Sigma') \in$ *br-step* δ
shows $\Sigma' \in$ *br-invar* δ
⟨*proof*⟩

lemma *br-invar-final*:

$\forall\Sigma.\ \Sigma\in wa\text{-}invar\ (br\text{-}algo\ \delta) \wedge \Sigma\notin wa\text{-}cond\ (br\text{-}algo\ \delta)$
$\longrightarrow fst\ \Sigma = b\text{-}accessible\ \delta$
⟨*proof*⟩

theorem *br-while-algo*:
assumes *FIN*[*simp*]: *finite* δ
shows *while-algo* (*br-algo* δ)
⟨*proof*⟩

lemma *bre-invar-final*:
$\forall\Sigma.\ \Sigma\in wa\text{-}invar\ (bre\text{-}algo\ Qi\ \delta) \wedge \Sigma\notin wa\text{-}cond\ (bre\text{-}algo\ Qi\ \delta)$
$\longrightarrow ((Qi\cap fst\ \Sigma=\{\}) \longleftrightarrow (Qi \cap b\text{-}accessible\ \delta = \{\}))$
⟨*proof*⟩

theorem *bre-while-algo*:
assumes *FIN*[*simp*]: *finite* δ
shows *while-algo* (*bre-algo Qi* δ)
⟨*proof*⟩

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 $('Q,'L)\ br'\text{-}state = 'Q\ set \times 'Q\ set \times (('Q,'L)\ ta\text{-}rule \rightharpoonup nat)$

— Abstraction to α-level
definition $br'\text{-}\alpha :: ('Q,'L)\ br'\text{-}state \Rightarrow ('Q,'L)\ br\text{-}state$
where $br'\text{-}\alpha = (\lambda(Q,W,rcm).\ (Q,W))$

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

definition $br'\text{-}invar :: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow ('Q,'L)\ br'\text{-}state\ set$
where $br'\text{-}invar\ \delta == br'\text{-}invar\text{-}add\ \delta \cap \{\Sigma.\ br'\text{-}\alpha\ \Sigma \in br\text{-}invar\ \delta\}$

inductive-set *br′-step*
$:: ('Q,'L)\ ta\text{-}rule\ set \Rightarrow (('Q,'L)\ br'\text{-}state \times ('Q,'L)\ br'\text{-}state)\ set$
for δ **where**
$[\![\ q\in W;$

$Q' = Q \cup \{ \; lhs \; r \mid r. \; r{\in}\delta \wedge q \in set \; (rhsq \; r) \wedge the \; (rcm \; r) \leq 1 \; \}$;
$W' = (W{-}\{q\})$
$\cup \; (\{ \; lhs \; r \mid r. \; r{\in}\delta \wedge q \in set \; (rhsq \; r) \wedge the \; (rcm \; r) \leq 1 \; \}$
$- \; Q)$;
$!!r. \; r{\in}\delta \Longrightarrow rcm' \; r = (\; if \; q \in set \; (rhsq \; r) \; then$
$Some \; (the \; (rcm \; r) - 1)$
$else \; rcm \; r$
$)$
$\rrbracket \Longrightarrow ((Q,W,rcm),(Q',W',rcm')) \in br'\text{-}step \; \delta$

definition *br′-cond* :: (′*Q*,′*L*) *br′-state set*
where $br'\text{-}cond == \{(Q,W,rcm). \; W{\neq}\{\}\}$
definition *bre′-cond* :: ′*Q set* ⇒ (′*Q*,′*L*) *br′-state set*
where $bre'\text{-}cond \; Qi == \{(Q,W,rcm). \; W{\neq}\{\} \wedge (Qi{\cap}Q{=}\{\})\}$

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

definition $br'\text{-}algo \; \delta == (\!|$
wa-cond=*br′-cond*,
wa-step = *br′-step* δ,
wa-initial = *br′-initial* δ,
wa-invar = *br′-invar* δ
$|\!)$

definition $bre'\text{-}algo \; Qi \; \delta == (\!|$
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*: $\Sigma{\in}br'\text{-}invar\text{-}add \; \delta \quad br'\text{-}\alpha \; \Sigma \in br\text{-}invar \; \delta$
assumes *STEP*: $(\Sigma,\Sigma') \in br'\text{-}step \; \delta$
shows $\Sigma'{\in}br'\text{-}invar\text{-}add \; \delta$
⟨*proof*⟩

lemma *br′-invar-initial*:
$br'\text{-}initial \; \delta \subseteq br'\text{-}invar\text{-}add \; \delta$
⟨*proof*⟩

lemma *br′-rcm-aux′*:
$\llbracket \; (Q,W,rcm){\in}br'\text{-}invar \; \delta; \; q{\in}W \; \rrbracket$
$\Longrightarrow \{r \in \delta. \; q \in set \; (rhsq \; r) \wedge the \; (rcm \; r) \leq Suc \; 0\}$
$= \{r{\in}\delta. \; q{\in}set \; (rhsq \; r) \wedge set \; (rhsq \; r) \subseteq (Q - (W{-}\{q\}))\}$

⟨*proof*⟩

lemma *br′-rcm-aux*:
assumes *A*: $(Q, W, rcm) \in br'\text{-}invar\ \delta$ $q \in W$
shows $\{lhs\ r\ |r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$
$= \{lhs\ r\ |\ r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge set\ (rhsq\ r) \subseteq (Q - (W - \{q\}))\}$
⟨*proof*⟩

lemma *br′-invar-QcD*:
$(Q, W, rcm) \in br'\text{-}invar\ \delta \Longrightarrow \{lhs\ r\ |\ r.\ r \in \delta \wedge set\ (rhsq\ r) \subseteq (Q - W)\} \subseteq Q$
⟨*proof*⟩

lemma *br′-rcm-aux2*:
$[\![\ (Q, W, rcm) \in br'\text{-}invar\ \delta;\ q \in W\ ]\!]$
$\Longrightarrow Q \cup br\text{-}dsq\ \delta\ q\ (Q, W)$
$= Q \cup \{lhs\ r\ |r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$
⟨*proof*⟩

lemma *br′-rcm-aux3*:
$[\![\ (Q, W, rcm) \in br'\text{-}invar\ \delta;\ q \in W\ ]\!]$
$\Longrightarrow br\text{-}dsq\ \delta\ q\ (Q, W) - Q$
$= \{lhs\ r\ |r.\ r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\} - Q$
⟨*proof*⟩

lemma *br′-step-abs*:
$[\![$
$\Sigma \in br'\text{-}invar\ \delta;$
$(\Sigma, \Sigma') \in br'\text{-}step\ \delta$
$]\!] \Longrightarrow (br'\text{-}\alpha\ \Sigma,\ br'\text{-}\alpha\ \Sigma') \in br\text{-}step\ \delta$
⟨*proof*⟩

lemma *br′-initial-abs*: $br'\text{-}\alpha\ `(br'\text{-}initial\ \delta) = \{br\text{-}initial\ \delta\}$
⟨*proof*⟩

lemma *br′-cond-abs*: $\Sigma \in br'\text{-}cond \longleftrightarrow (br'\text{-}\alpha\ \Sigma) \in br\text{-}cond$
⟨*proof*⟩

lemma *bre′-cond-abs*: $\Sigma \in bre'\text{-}cond\ Qi \longleftrightarrow (br'\text{-}\alpha\ \Sigma) \in bre\text{-}cond\ Qi$
⟨*proof*⟩

lemma *br′-invar-abs*: $br'\text{-}\alpha\ `br'\text{-}invar\ \delta \subseteq br\text{-}invar\ \delta$
⟨*proof*⟩

theorem *br′-pref-br*: *wa-precise-refine* $(br'\text{-}algo\ \delta)\ (br\text{-}algo\ \delta)\ br'\text{-}\alpha$
⟨*proof*⟩

interpretation *br′-pref*: *wa-precise-refine* $br'\text{-}algo\ \delta$ $br\text{-}algo\ \delta$ $br'\text{-}\alpha$
⟨*proof*⟩

theorem *br′-while-algo*:
  finite $\delta \Longrightarrow$ *while-algo* (*br′-algo* δ)
  ⟨*proof*⟩

lemma *fst-br′-*α: *fst* (*br′-*α *s*) = *fst s* ⟨*proof*⟩

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
  ⟨*proof*⟩

interpretation *bre′-pref*:
  wa-precise-refine bre′-algo Qi δ *bre-algo Qi* δ *br′-*α
  ⟨*proof*⟩

theorem *bre′-while-algo*:
  finite $\delta \Longrightarrow$ *while-algo* (*bre′-algo Qi* δ)
  ⟨*proof*⟩

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*:

$\llbracket$ *q*∈*W*; *br'-inner-invar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*) {} Σ' $\rrbracket$
  ⟹ ((*Q*,*W*,*rcm*),Σ') ∈ *br'-step δ*
⟨*proof*⟩

lemma *br'-inner-invar-step*:
$\llbracket$ *q*∈*W*; *br'-inner-invar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*) *it* Σ';
  r∈*it*; *it*⊆{*r*∈*δ*. *q*∈*set* (*rhsq r*)}
$\rrbracket$ ⟹ *br'-inner-invar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*)
  (*it*−{*r*}) (*br'-inner-step r* Σ')

⟨*proof*⟩

lemma *br'-inner-invar-initial*:
$\llbracket$ *q*∈*W* $\rrbracket$ ⟹ *br'-inner-invar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*)
  {*r*∈*δ*. *q*∈*set* (*rhsq r*)} (*Q*,*W*−{*q*},*rcm*)
⟨*proof*⟩

lemma *br'-inner-step-proof*:
fixes *αs* :: 'Σ ⇒ ('*Q*,'*L*) *br'-state*
fixes *cstep* :: ('*Q*,'*L*) *ta-rule* ⇒ 'Σ ⇒ 'Σ
fixes Σ*h* :: 'Σ
fixes *cinvar* :: ('*Q*,'*L*) *ta-rule set* ⇒ 'Σ ⇒ *bool*

assumes *iterable-set*: *set-iteratei α invar iteratei*
assumes *invar-initial*: *cinvar* {*r*∈*δ*. *q*∈*set* (*rhsq r*)} Σ*h*
assumes *invar-step*:
  !!*it r* Σ. $\llbracket$ *r*∈*it*; *it* ⊆ {*r*∈*δ*. *q*∈*set* (*rhsq r*)}; *cinvar it* Σ $\rrbracket$
    ⟹ *cinvar* (*it*−{*r*}) (*cstep r* Σ)
assumes *step-desc*:
  !!*it r* Σ. $\llbracket$ *r*∈*it*; *it*⊆{*r*∈*δ*. *q*∈*set* (*rhsq r*)}; *cinvar it* Σ $\rrbracket$
    ⟹ *αs* (*cstep r* Σ) = *br'-inner-step r* (*αs* Σ)
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 Σ*h-desc*[*simp*]: *αs* Σ*h* = (*Q*,*W*−{*q*},*rcm*)

shows (*αs* Σ, *αs* (*iteratei it-set* (λ-. *True*) *cstep* Σ*h*))∈*br'-step δ*
⟨*proof*⟩

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.

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*
$:: ('Q \rightharpoonup 'L\ tree) \Rightarrow ('Q,'L)\ ta\text{-}rule \Rightarrow 'L\ tree$
where
$construct\text{-}witness\ Q\ r == NODE\ (rhsl\ r)\ (List.map\ (\lambda q.\ the\ (Q\ q))\ (rhsq\ r))$

lemma *witness-propD*: $[\![witness\text{-}prop\ \delta\ m;\ m\ q = Some\ t]\!] \Longrightarrow accs\ \delta\ t\ q$
⟨*proof*⟩

lemma *construct-witness-correct*:
$[\![\ witness\text{-}prop\ \delta\ Q;\ r{\in}\delta;\ set\ (rhsq\ r) \subseteq dom\ Q\ ]\!]$
$\Longrightarrow accs\ \delta\ (construct\text{-}witness\ Q\ r)\ (lhs\ r)$
⟨*proof*⟩

lemma *construct-witness-eq*:
$[\![\ Q\ |`\ set\ (rhsq\ r) = Q'\ |`\ set\ (rhsq\ r)\ ]\!] \Longrightarrow$
$construct\text{-}witness\ Q\ r = construct\text{-}witness\ Q'\ r$
⟨*proof*⟩

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

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

definition $brw\text{-}\alpha :: ('Q,'L)\ brw\text{-}state \Rightarrow ('Q,'L)\ br'\text{-}state$
where $brw\text{-}\alpha = (\lambda(Q,W,rcm).\ (dom\ Q,W,rcm))$

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

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

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

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

inductive-set *brw-iq* :: (*'Q*,*'L*) *ta-rule set* ⇒ (*'Q* ⇀ *'L tree*) *set*
for *δ* **where**
⟦
∀ *q t*. *Q q* = *Some t* ⟶ (∃ *r*∈*δ*. *rhsq r* = [] ∧ *q* = *lhs r*
∧ *t* = *NODE* (*rhsl r*) []);
∀ *r*∈*δ*. *rhsq r* = [] ⟶ *Q* (*lhs r*) ≠ *None*
⟧ ⟹ *Q* ∈ *brw-iq δ*

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

definition *brw-algo Qi δ* == ⦇
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*
⟨*proof*⟩

lemma *brw-initial-abs*: Σ∈*brw-initial δ* ⟹ *brw-α* Σ ∈ *br'-initial δ*
⟨*proof*⟩

lemma *brw-invar-initial*: *brw-initial δ* ⊆ *brw-invar-add δ*
⟨*proof*⟩

lemma *brw-step-abs*:
⟦ (Σ,Σ')∈*brw-step δ* ⟧ ⟹ (*brw-α* Σ, *brw-α* Σ')∈*br'-step δ*
⟨*proof*⟩

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*⟩

theorem *brw-pref-bre'*: *wa-precise-refine* (*brw-algo Qi δ*) (*bre'-algo Qi δ*) *brw-α*
⟨*proof*⟩

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

⟨*proof*⟩

theorem *brw-while-algo*: *finite* $\delta \Longrightarrow$ *while-algo* (*brw-algo Qi* δ)
⟨*proof*⟩

lemma *fst-brw-*α: *fst* (*brw-*α *s*) = *dom* (*fst s*)
⟨*proof*⟩

theorem *brw-invar-final*:
$\forall$ *sc. sc* $\in$ *wa-invar* (*brw-algo Qi* δ) $\wedge$ *sc* $\notin$ *wa-cond* (*brw-algo Qi* δ)
$\longrightarrow$ (*Qi* $\cap$ *dom* (*fst sc*) = {}) = (*Qi* $\cap$ *b-accessible* δ = {})
$\wedge$ (*witness-prop* δ (*fst sc*))
⟨*proof*⟩

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*
⇒ (*'Q*,*'L*) *brw-state* ⇒ *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*′)) ∧
(*Q*′|‘*dom Q* = *Q*) ∧
(*let dsqr* = { *r*∈*rules* − *it*. *the* (*rcm r*) ≤ *1* } *in*
(∀ *q t*. *Q*′ *q* = *Some t* ⟶ (*Q q* = *Some t*
∨ (*Q q* = *None* ∧ (∃ *r*∈*dsqr*. *q*=*lhs r* ∧ *t*=*construct-witness Q r*))
)
)))

lemma *brw-inner-step-abs*:
(Σ,Σ′)∈*brw-inner-step r* ⟹ *br*′*-inner-step r* (*brw-*α Σ) = *brw-*α Σ′
⟨*proof*⟩

lemma *brw-inner-invar-imp-final*:
⟦ *q*∈*W*; *brw-inner-invar* {*r*∈δ. *q*∈*set* (*rhsq r*)} *q* (*Q*,*W*−{*q*},*rcm*) {} Σ′ ⟧
⟹ ((*Q*,*W*,*rcm*),Σ′) ∈ *brw-step* δ

⟨*proof*⟩

lemma *brw-inner-invar-step*:
assumes *INVI*: $(Q, W, rcm) \in brw\text{-}invar\ \delta$
assumes *A*: $q \in W \quad r \in it \quad it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\}$
assumes *INVH*: $brw\text{-}inner\text{-}invar\ \{r \in \delta.\ q \in set\ (rhsq\ r)\}\ q\ (Q, W - \{q\}, rcm)\ it\ \Sigma h$
assumes *STEP*: $(\Sigma h, \Sigma') \in brw\text{-}inner\text{-}step\ r$
shows $brw\text{-}inner\text{-}invar\ \{r \in \delta.\ q \in set\ (rhsq\ r)\}\ q\ (Q, W - \{q\}, rcm)\ (it - \{r\})\ \Sigma'$
⟨*proof*⟩

lemma *brw-inner-invar-initial*:
$[\![q \in W]\!] \Longrightarrow brw\text{-}inner\text{-}invar\ \{r \in \delta.\ q \in set\ (rhsq\ r)\}\ q\ (Q, W - \{q\}, rcm)$
$\{r \in \delta.\ q \in set\ (rhsq\ r)\}\ (Q, W - \{q\}, rcm)$
⟨*proof*⟩

theorem *brw-inner-step-proof*:
fixes $\alpha s :: {}'\Sigma \Rightarrow ({}'Q, {}'L)\ brw\text{-}state$
fixes $cstep :: ({}'Q, {}'L)\ ta\text{-}rule \Rightarrow {}'\Sigma \Rightarrow {}'\Sigma$
fixes $\Sigma h :: {}'\Sigma$
fixes $cinvar :: ({}'Q, {}'L)\ ta\text{-}rule\ set \Rightarrow {}'\Sigma \Rightarrow bool$

assumes *set-iterate*: *set-iteratei* α *invar iteratei*
assumes *invar-start*: $(\alpha s\ \Sigma) \in brw\text{-}invar\ \delta$
assumes *invar-initial*: $cinvar\ \{r \in \delta.\ q \in set\ (rhsq\ r)\}\ \Sigma h$
assumes *invar-step*:
$!!it\ r\ \Sigma.\ [\![\ r \in it;\ it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\};\ cinvar\ it\ \Sigma\ ]\!]$
$\Longrightarrow cinvar\ (it - \{r\})\ (cstep\ r\ \Sigma)$
assumes *step-desc*:
$!!it\ r\ \Sigma.\ [\![\ r \in it;\ it \subseteq \{r \in \delta.\ q \in set\ (rhsq\ r)\};\ cinvar\ it\ \Sigma\ ]\!]$
$\Longrightarrow (\alpha s\ \Sigma,\ \alpha s\ (cstep\ r\ \Sigma)) \in brw\text{-}inner\text{-}step\ r$
assumes *it-set-desc*: *invar it-set* $\quad \alpha\ it\text{-}set = \{r \in \delta.\ q \in set\ (rhsq\ r)\}$

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 $(\alpha s\ \Sigma,\ \alpha s\ (iteratei\ it\text{-}set\ (\lambda\text{-}.\ True)\ cstep\ \Sigma h)) \in brw\text{-}step\ \delta$
⟨*proof*⟩

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
⟨*proof*⟩

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 ⟨*proof*⟩
theorem *frp-while-algo*:
assumes *TA*: *tree-automaton T1*
tree-automaton T2
shows *while-algo* (*frp-algo T1 T2*)
⟨*proof*⟩
theorem *frp-inv-final*:
∀ *s*. *s*∈*wa-invar* (*frp-algo T1 T2*) ∧ *s*∉*wa-cond* (*frp-algo T1 T2*)
⟶ (*case s of* (*Q*,*W*,*δd*) ⇒
⦇ *ta-initial* = *ta-initial T1* × *ta-initial T2*,
ta-rules = *δd*
⦈ = *ta-fwd-reduce* (*ta-prod T1 T2*))
⟨*proof*⟩

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

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
⟨*proof*⟩
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 ⟨*proof*⟩

end

5.1.1 Ad-Hoc instantiations of generic Algorithms

⟨*ML*⟩
interpretation *hll-idx*: *build-index-loc hm-ops ls-ops ls-ops* ⟨*proof*⟩
interpretation *ll-set-xy*: *g-set-xy-loc ls-ops ls-ops*
⟨*proof*⟩

interpretation *lh-set-xx*: *g-set-xx-loc ls-ops hs-ops*
⟨*proof*⟩
interpretation *lll-iflt-cp*: *inj-image-filter-cp-loc ls-ops ls-ops ls-ops*
⟨*proof*⟩
interpretation *hhh-cart*: *cart-loc hs-ops hs-ops hs-ops* ⟨*proof*⟩
interpretation *hh-set-xy*: *g-set-xy-loc hs-ops hs-ops*
⟨*proof*⟩

interpretation *llh-set-xyy*: *g-set-xyy-loc ls-ops ls-ops hs-ops*
⟨*proof*⟩

interpretation *hh-map-to-nat*: *map-to-nat-loc hs-ops hm-ops* ⟨*proof*⟩
interpretation *hh-set-xy*: *g-set-xy-loc hs-ops hs-ops* ⟨*proof*⟩
interpretation *lh-set-xy*: *g-set-xy-loc ls-ops hs-ops* ⟨*proof*⟩
interpretation *hh-set-xx*: *g-set-xx-loc hs-ops hs-ops* ⟨*proof*⟩
interpretation *hs-to-fifo*: *set-to-list-loc hs-ops fifo-ops* ⟨*proof*⟩

⟨*ML*⟩

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* (λr. *rhsl r*) δ
definition *build-rule-index-s* δ == *build-rule-index* (λr. *lhs r*) δ
definition *build-rule-index-sf* δ == *build-rule-index* (λr. (*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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *build-rule-index-sf-correct*[*simp*]:
assumes *I*[*simp*, *intro*!]: *ls-invar* δ
shows
hll-idx.is-index (λr. (*lhs r*, *rhsl r*)) (*ls-α* δ) (*build-rule-index-sf* δ)
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *hta-ensure-idx-s-correct-α*[*simp*]:
hta-α (*hta-ensure-idx-s H*) = *hta-α H*
⟨*proof*⟩
lemma *hta-ensure-idx-sf-correct-α*[*simp*]:
hta-α (*hta-ensure-idx-sf H*) = *hta-α H*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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

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

$\Longrightarrow$ *hll-idx.is-index* ($\lambda r.$ (*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 $\Longrightarrow$ *ls-α* (*hta-lookup-f f H*) = $\{r \in$ *ls-α* δ . *rhsl r* = $f\}$
hta-has-idx-f H $\Longrightarrow$ *ls-invar* (*hta-lookup-f f H*)
⟨*proof*⟩

lemma *hta-lookup-s-correct*:
hta-has-idx-s H $\Longrightarrow$ *ls-α* (*hta-lookup-s q H*) = $\{r \in$ *ls-α* δ . *lhs r* = $q\}$
hta-has-idx-s H $\Longrightarrow$ *ls-invar* (*hta-lookup-s q H*)
⟨*proof*⟩

lemma *hta-lookup-sf-correct*:
hta-has-idx-sf H
$\Longrightarrow$ *ls-α* (*hta-lookup-sf q f H*) = $\{r \in$ *ls-α* δ . *lhs r* = $q \wedge$ *rhsl r* = $f\}$
hta-has-idx-sf H $\Longrightarrow$ *ls-invar* (*hta-lookup-sf q f H*)
⟨*proof*⟩
lemma *hta-ensure-idx-f-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-f H*)
⟨*proof*⟩

lemma *hta-ensure-idx-s-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-s H*)
⟨*proof*⟩

lemma *hta-ensure-idx-sf-correct*[*simp*, *intro*!]: *hashedTa* (*hta-ensure-idx-sf H*)
⟨*proof*⟩

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

lemma *hta-α-is-ta*[*simp*, *intro*!]: *tree-automaton* (*hta-α H*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩
lemma *hta-mem′-correct*:
hta-has-idx-f H ⟹ hta-mem′ t H ⟷ t∈ta-lang (hta-α H)
⟨*proof*⟩

theorem *hta-mem-correct*: *hta-mem t H ⟷ t∈ta-lang (hta-α H)*
⟨*proof*⟩

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
∧ 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′*
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

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".

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⟩
```

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*: $(Q,W,\delta d) \in$ *pa-invar H1 H2*
  assumes *COND*: *pa-cond* $(Q,W,\delta d)$
  shows
    (*pa-step H1 H2* $(Q,W,\delta d)) \in$ *pa-invar-add* (**is** *?T1*)
    (*pa-*α $(Q,W,\delta d)$, *pa-*α (*pa-step H1 H2* $(Q,W,\delta d)$))
     $\in$ *frp-step* (*ls-*α (*hta-*δ *H1*)) (*ls-*α (*hta-*δ *H2*)) (**is** *?T2*)
$\langle proof \rangle$
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
$\langle proof \rangle$
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*))
$\langle proof \rangle$
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 $(Q,W,\delta d)$ = *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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

5.6 Remap States

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*)
⟨*proof*⟩

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.

fun *rule-states-l* **where**
rule-states-l $(q \to f\ qs)$ = *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*)
⟨*proof*⟩

definition *hta-δ-states H*
== (*llh-set-xyy.g-Union-image id* (*ll-set-xy.g-image-filter*
(λ*r. 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*⟩

lemma (**in** *hashedTa*) *hta-states-correct*:
hs-α (*hta-states H*) = *ta-rstates* (*hta-α H*)
hs-invar (*hta-states H*)
⟨*proof*⟩

definition *reindex-map H* ==
λ*q. 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 (λ*q. the* (*hm-lookup q mp*)) *H*)

⟨*proof*⟩

lemma (**in** *hashedTa*) *reindex-map-correct*:
inj-on (*reindex-map H*) (*ta-rstates* (*hta-α H*))
⟨*proof*⟩

theorem (**in** *hashedTa*) *hta-reindex-correct*:
ta-lang (*hta-α* (*hta-reindex H*)) = *ta-lang* (*hta-α H*)
hashedTa (*hta-reindex H*)

⟨*proof*⟩

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*⟩

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*⟩

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.

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* ())
⟨*proof*⟩
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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
    (λ*r*. *if hs-memb* (*lhs r*) *Q* ∧ *list-all* (λ*q*. *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*)
⟨*proof*⟩

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.

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*)
(λ*r res*.
foldl (λ*res 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*′)
)
⟨*proof*⟩

lemma *rqrm-propD*:
rqrm-prop δ *rqrm* ⟹ *ls*-α (*rqrm-lookup rqrm q*) = {*r*∈δ. *q*∈*set* (*rhsq r*)}
⟨*proof*⟩

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*⟩
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*)
(λ*r 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*
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩

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*))
)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *brc-initial-invar*: *ls-invar* δ ⟹ (*brc-initial* δ)∈*brc-invar-add*
⟨*proof*⟩

lemma *brc-cond-abs*: *brc-cond* Σ ⟷ (*brc-α* Σ)∈*br′-cond*
⟨*proof*⟩

lemma *brc-initial-abs*:
ls-invar δ ⟹ *brc-α* (*brc-initial* δ) ∈ *br′-initial* (*ls-α* δ)
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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 $\cap$ $\{(Q,W,\text{-},\text{-})$. *set* (*fifo-α W*) $\subseteq$ *dom* (*hm-α Q*)$\}$

lemma *brec-invar-cons*:
$\Sigma\in$*brec-invar Qi δ* $\Longrightarrow$ $\Sigma\in$*brec-invar-inner Qi*
⟨*proof*⟩

lemma *brec-brw-invar-cons*:
brec-α Σ $\in$ *brw-invar Qi* $\Longrightarrow$ *set* (*fifo-α* (*fst* (*snd* Σ))) $\subseteq$ *dom* (*hm-α* (*fst* Σ))
⟨*proof*⟩

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*$\in$*ls-α δ* $\wedge$ *rhsq r* = []$\}$ (**is** *?T1*)
witness-prop (*ls-α δ*) (*hm-α* (*brec-iqm δ*)) (**is** *?T2*)
hm-invar (*brec-iqm δ*) (**is** *?T3*)
⟨*proof*⟩

lemma *brec-iqm-correct*:
assumes *INV*[*simp*]: *ls-invar δ*
shows *hm-α* (*brec-iqm δ*) $\in$ *brw-iq* (*ls-α δ*)
⟨*proof*⟩

lemma *brec-inner-step-brw-desc*:
⟦ $\Sigma\in$*brec-invar-inner* (*hs-α Qi*) ⟧
$\Longrightarrow$ (*brec-α* Σ, *brec-α* (*brec-inner-step Qi r* Σ)) $\in$ *brw-inner-step r*
⟨*proof*⟩

lemma *brec-step-invar*:
assumes *RQRM*: *rqrm-invar rqrm* *rqrm-prop δ rqrm*
assumes [*simp*]: *hs-invar Qi*
shows ⟦ $\Sigma\in$*brec-invar-add* (*hs-α Qi*); *brec-α* Σ $\in$ *brw-invar δ*; *brec-cond* Σ ⟧
$\Longrightarrow$ (*brec-step rqrm Qi* Σ)$\in$*brec-invar-add* (*hs-α Qi*)
⟨*proof*⟩

lemma *brec-step-abs*:
assumes *RQRM*: *rqrm-invar rqrm* *rqrm-prop δ rqrm*
assumes *INV*[*simp*]: *hs-invar Qi*
assumes *A′*: $\Sigma\in$*brec-invar* (*hs-α Qi*) *δ*
assumes *COND*: *brec-cond* Σ
shows (*brec-α* Σ, *brec-α* (*brec-step rqrm Qi* Σ)) $\in$ *brw-step δ*

⟨*proof*⟩

lemma *brec-invar-initial*:
  $[\![$*ls-invar* δ; *hs-invar Qi*$]\!] \Longrightarrow$ (*brec-initial Qi* δ) $\in$ *brec-invar-add* (*hs-*α *Qi*)
  ⟨*proof*⟩

lemma *brec-cond-abs*:
  $[\![\Sigma\in$*brec-invar Qi* $\delta]\!] \Longrightarrow$ *brec-cond* $\Sigma \longleftrightarrow$ (*brec-*α Σ)$\in$*brw-cond Qi*
  ⟨*proof*⟩

lemma *brec-initial-abs*:
  $[\![$ *ls-invar* δ; *hs-invar Qi* $]\!]$
    $\Longrightarrow$ *brec-*α (*brec-initial Qi* δ) $\in$ *brw-initial* (*ls-*α δ)
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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*⟩

lemma *fst-brec-*α: *fst* (*brec-*α Σ) = *hm-*α (*fst* Σ)
  ⟨*proof*⟩

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*⟩

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* ())
  ⟨*proof*⟩

  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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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* ⟹ *hta-α* (*hta-add-qi qi H*) = ⦇*ta-initial* = *insert qi* (*ta-initial* (*hta-α H*)), *ta-rules* = *ta-rules* (*hta-α H*)⦈

hashedTa H ⟹ *hashedTa* (*hta-add-qi qi H*)

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

Relevant Lemmas

hashedTa.hta-add-rule-correct: *hashedTa H* ⟹ *hta-α* (*hta-add-rule r H*) = ⦇*ta-initial* = *ta-initial* (*hta-α H*), *ta-rules* = *insert r* (*ta-rules* (*hta-α H*))⦈

hashedTa H ⟹ *hashedTa* (*hta-add-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: $\textit{hashedTa}\ H \Longrightarrow \textit{hta-mem}\ t\ H = (t \in \textit{ta-lang}\ (\textit{hta-}\alpha\ H))$

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

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

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

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

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

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

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

hta-prod′-correct: $[\![$*hashedTa H1*; *hashedTa H2*; *hta-has-idx-s H1*; *hta-has-idx-sf H2*$]\!] \Longrightarrow$ *ta-lang* (*hta-α* (*hta-prod′ H1 H2*)) = *ta-lang* (*hta-α H1*) ∩ *ta-lang* (*hta-α H2*)

$[\![$*hashedTa H1*; *hashedTa H2*; *hta-has-idx-s H1*; *hta-has-idx-sf H2*$]\!] \Longrightarrow$ *hashedTa* (*hta-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-α* (*hta-prodWR H1 H2*) = *ta-prod* (*hta-α H1*) (*hta-α H2*)

$[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *hashedTa* (*hta-prodWR H1 H2*)

hta-prodWR-correct: $[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *ta-lang* (*hta-α* (*hta-prodWR H1 H2*)) = *ta-lang* (*hta-α H1*) ∩ *ta-lang* (*hta-α H2*)

$[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *hashedTa* (*hta-prodWR H1 H2*)

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

Relevant Lemmas

hta-union-correct′: $[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *hta-α* (*hta-union H1 H2*) = *ta-union-wrap* (*hta-α H1*) (*hta-α H2*)

$[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *hashedTa* (*hta-union H1 H2*)

hta-union-correct: $[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *ta-lang* (*hta-α* (*hta-union H1 H2*)) = *ta-lang* (*hta-α H1*) ∪ *ta-lang* (*hta-α H2*)

$[\![$*hashedTa H1*; *hashedTa H2*$]\!] \Longrightarrow$ *hashedTa* (*hta-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*⟩

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.