

Tree Automata

Peter Lammich

December 7, 2022

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	4
1.1	Submission Structure	4
1.1.1	common/	4
1.1.2	common/bugfixes/	5
1.1.3	./	5
1.1.4	code/	5
1.1.5	code/ml/	6
1.1.6	code/ocaml/	6
1.1.7	code/haskell/	6
1.1.8	code/taml/	7
2	Trees	7
3	Tree Automata	7
3.1	Basic Definitions	8
3.1.1	Tree Automata	8
3.1.2	Acceptance	8
3.1.3	Language	9
3.2	Basic Properties	9
3.3	Other Classes of Tree Automata	11
3.3.1	Automata over Ranked Alphabets	11
3.3.2	Deterministic Tree Automata	12
3.3.3	Complete Tree Automata	12
3.4	Algorithms	12
3.4.1	Empty Automaton	13
3.4.2	Remapping of States	13
3.4.3	Union	14
3.4.4	Reduction	16
3.4.5	Product Automaton	19
3.4.6	Determinization	21
3.4.7	Completion	23
3.4.8	Complement	23
3.5	Regular Tree Languages	24
3.5.1	Definitions	24
3.5.2	Closure Properties	25
4	Abstract Tree Automata Algorithms	26
4.1	Word Problem	26
4.2	Backward Reduction and Emptiness Check	27
4.2.1	Auxiliary Definitions	27
4.2.2	Algorithms	27
4.3	Product Automaton	38

5	Executable Implementation of Tree Automata	40
5.1	Prelude	40
5.1.1	Ad-Hoc instantiations of generic Algorithms	41
5.2	Generating Indices of Rules	42
5.3	Tree Automaton with Optional Indices	42
5.4	Algorithm for the Word Problem	46
5.5	Product Automaton and Intersection	47
5.5.1	Brute Force Product Automaton	47
5.5.2	Product Automaton with Forward-Reduction	48
5.6	Remap States	51
5.6.1	Reindex Automaton	51
5.7	Union	53
5.8	Operators to Construct Tree Automata	53
5.9	Backwards Reduction and Emptiness Check	54
5.9.1	Emptiness Check with Witness Computation	59
5.10	Interface for Natural Number States and Symbols	63
5.11	Interface Documentation	65
5.11.1	Building a Tree Automaton	65
5.11.2	Basic Operations	66
5.12	Code Generation	68
6	Conclusion	70
6.1	Efficiency of Generated Code	70
6.2	Future Work	71
6.3	Trusted Code Base	71

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 were 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, reduction¹, 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.

¹Currently 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 finished, 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 $q \rightarrow l \ q1 \dots \ qn$, with the meaning that one can derive $l(q1 \dots \ qn)$ 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* ($q \rightarrow l \ qs$) = q

fun *rhsq* **where** *rhsq* ($q \rightarrow l \ qs$) = qs

fun *rhsl* **where** *rhsl* ($q \rightarrow l \ qs$) = l

— States in a rule

fun *rule-states* **where** *rule-states* ($q \rightarrow l \ qs$) = *insert* q (*set* qs)

— States in a set of rules

definition *δ-states* δ == \bigcup (*rule-states* ' δ)

— States in a tree automaton

definition *ta-rstates* TA = *ta-initial* $TA \cup \delta$ -*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 $accs :: ('Q, 'L) \text{ ta-rule set} \Rightarrow 'L \text{ tree} \Rightarrow 'Q \Rightarrow \text{bool}$
where

\llbracket
 $(q \rightarrow f \text{ qs}) \in \delta; \text{length } ts = \text{length } qs;$
 $!!i. i < \text{length } qs \implies accs \delta (ts ! i) (qs ! i)$
 $\rrbracket \implies accs \delta (NODE f ts) q$

— Characterization of $accs$ using $list\text{-all-}zip$

inductive $accs\text{-laz} :: ('Q, 'L) \text{ ta-rule set} \Rightarrow 'L \text{ tree} \Rightarrow 'Q \Rightarrow \text{bool}$
where

\llbracket
 $(q \rightarrow f \text{ qs}) \in \delta;$
 $list\text{-all-}zip (accs\text{-laz } \delta) ts \text{ qs}$
 $\rrbracket \implies accs\text{-laz } \delta (NODE f ts) q$

lemma $accs\text{-laz}: accs = accs\text{-laz}$
 $\langle proof \rangle$

3.1.3 Language

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

definition $ta\text{-lang } TA == \{ t . \exists q \in ta\text{-initial } TA. accs (ta\text{-rules } TA) t q \}$

3.2 Basic Properties

lemma $rule\text{-states}\text{-simp}$:

$rule\text{-states } x = (\text{case } x \text{ of } (q \rightarrow l \text{ qs}) \Rightarrow \text{insert } q (\text{set } qs))$
 $\langle proof \rangle$

lemma $rule\text{-states}\text{-lhs}[simp]$: $lhs \ r \in rule\text{-states } r$
 $\langle proof \rangle$

lemma $rule\text{-states}\text{-rhsq}$: $set (rhsq \ r) \subseteq rule\text{-states } r$
 $\langle proof \rangle$

lemma $rule\text{-states}\text{-finite}[simp, intro!]$: $finite (rule\text{-states } r)$
 $\langle proof \rangle$

lemma $\delta\text{-states}I$:

assumes $A: (q \rightarrow l \text{ qs}) \in \delta$
shows $q \in \delta\text{-states } \delta$
 $set \text{ qs} \subseteq \delta\text{-states } \delta$
 $\langle proof \rangle$

lemma $\delta\text{-states}I'$: $\llbracket (q \rightarrow l \text{ qs}) \in \delta; qi \in set \text{ qs} \rrbracket \implies qi \in \delta\text{-states } \delta$
 $\langle proof \rangle$

lemma δ -states-accsI: $\text{accs } \delta \ n \ q \implies q \in \delta\text{-states } \delta$
<proof>

lemma δ -states-union[simp]: $\delta\text{-states } (\delta \cup \delta') = \delta\text{-states } \delta \cup \delta\text{-states } \delta'$
<proof>

lemma δ -states-insert[simp]:
 $\delta\text{-states } (\text{insert } r \ \delta) = (\text{rule-states } r \cup \delta\text{-states } \delta)$
<proof>

lemma δ -states-mono: $\llbracket \delta \subseteq \delta' \rrbracket \implies \delta\text{-states } \delta \subseteq \delta\text{-states } \delta'$
<proof>

lemma δ -states-finite[simp, intro]: $\text{finite } \delta \implies \text{finite } (\delta\text{-states } \delta)$
<proof>

lemma δ -statesE: $\llbracket q \in \delta\text{-states } \Delta; \$
 $\llbracket f \ qs. \llbracket (q \rightarrow f \ qs) \in \Delta \rrbracket \implies P; \$
 $\llbracket ql \ f \ qs. \llbracket (ql \rightarrow f \ qs) \in \Delta; q \in \text{set } qs \rrbracket \implies P \$
 $\rrbracket \implies P$
<proof>

lemma δ -symbolsI: $(q \rightarrow f \ qs) \in \delta \implies f \in \delta\text{-symbols } \delta$
<proof>

lemma δ -symbolsE:
 assumes $A: f \in \delta\text{-symbols } \delta$
 obtains $q \ qs$ **where** $(q \rightarrow f \ qs) \in \delta$
<proof>

lemma δ -symbols-simps[simp]:
 $\delta\text{-symbols } \{\} = \{\}$
 $\delta\text{-symbols } (\text{insert } r \ \delta) = \text{insert } (\text{rhdl } r) (\delta\text{-symbols } \delta)$
 $\delta\text{-symbols } (\delta \cup \delta') = \delta\text{-symbols } \delta \cup \delta\text{-symbols } \delta'$
<proof>

lemma δ -symbols-finite[simp, intro!]:
 $\text{finite } \delta \implies \text{finite } (\delta\text{-symbols } \delta)$
<proof>

lemma accs-mono: $\llbracket \text{accs } \delta \ n \ q; \delta \subseteq \delta' \rrbracket \implies \text{accs } \delta' \ n \ q$
<proof>

context *tree-automaton*

begin

lemma *initial-subset*: $\text{ta-initial } TA \subseteq \text{ta-rstates } TA$
 <proof>

lemma *states-subset*: $\delta\text{-states } (\text{ta-rules } TA) \subseteq \text{ta-rstates } TA$
 <proof>

```

lemma finite-states[simp, intro!]: finite (ta-rstates TA)
  <proof>

lemma finite-symbols[simp, intro!]: finite ( $\delta$ -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  $\rightarrow$  nat)  $\Rightarrow$  'L tree set
  for A where
  [[  $\forall t \in \text{set } ts. t \in \text{ranked-trees } A; A f = \text{Some } (\text{length } ts)$  ]]
   $\implies$  NODE f ts  $\in$  ranked-trees A

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

context finite-alphabet
begin

  definition legal-rules Q == { (q  $\rightarrow$  f qs) | q f qs.
    q  $\in$  Q
     $\wedge$  qs  $\in$  lists Q
     $\wedge$  A f = Some (length qs) }

  lemma legal-rulesI:
  [[
    r  $\in$   $\delta$ ;
    rule-states r  $\subseteq$  Q;
    A (rhsl r) = Some (length (rhsq r))
  ]]  $\implies$  r  $\in$  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  $\rightarrow$  nat +
  assumes ranked: (q  $\rightarrow$  f qs) $\in\delta \implies A f = \text{Some } (\text{length } qs)$ 
begin

  lemma rules-legal: r $\in\delta \implies r \in \text{legal-rules } Q$ 
  <proof>
  lemma accs-is-ranked: accs  $\delta t q \implies t \in \text{ranked-trees } A$ 
  <proof>
  theorem lang-is-ranked: ta-lang TA  $\subseteq$  ranked-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:  $\llbracket (q \rightarrow f qs) \in \delta; (q' \rightarrow f qs) \in \delta \rrbracket \implies q=q'$ 
begin
  theorem accs-unique:  $\llbracket \text{accs } \delta t q; \text{accs } \delta t q' \rrbracket \implies q=q'$ 
  <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:
   $\llbracket qs \in \text{lists } Q; A f = \text{Some } (\text{length } qs) \rrbracket \implies \exists q. (q \rightarrow f qs) \in \delta$ 
begin

```

— In a complete DFTA, all trees can be labeled by some state

```

theorem label-all: t $\in\text{ranked-trees } A \implies \exists q \in Q. \text{accs } \delta t q$ 
<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 $ta_empty == \langle \mid ta_initial = \{\}, ta_rules = \{\} \rangle$

theorem $ta_empty_lang[simp]$: $ta_lang\ ta_empty = \{\}$
 $\langle proof \rangle$

theorem $ta_empty_ta[simp, intro!]$: $tree_automaton\ ta_empty$
 $\langle proof \rangle$

theorem (in $finite_alphabet$) $ta_empty_rta[simp, intro!]$:
 $ranked_tree_automaton\ ta_empty\ A$
 $\langle proof \rangle$

theorem (in $finite_alphabet$) $ta_empty_dta[simp, intro!]$:
 $det_tree_automaton\ ta_empty\ A$
 $\langle proof \rangle$

3.4.2 Remapping of States

fun $remap_rule$ **where** $remap_rule\ f\ (q \rightarrow l\ qs) = ((f\ q) \rightarrow l\ (map\ f\ qs))$

definition

$$ta_remap\ f\ TA == \langle \mid ta_initial = f\ ' ta_initial\ TA,$$

$$ta_rules = remap_rule\ f\ ' ta_rules\ TA$$

$$\rangle$$

lemma $\delta_states_remap[simp]$: $\delta_states\ (remap_rule\ f\ ' \delta) = f\ ' \delta_states\ \delta$
 $\langle proof \rangle$

lemma $remap_accs1$: $accs\ \delta\ n\ q \implies accs\ (remap_rule\ f\ ' \delta)\ n\ (f\ q)$
 $\langle proof \rangle$

lemma $remap_lang1$: $t \in ta_lang\ TA \implies t \in ta_lang\ (ta_remap\ f\ TA)$
 $\langle proof \rangle$

lemma $remap_accs2$: \llbracket

$$accs\ \delta'\ n\ q';$$

$$\delta' = (remap_rule\ f\ ' \delta);$$

$$q' = f\ q;$$

$$inj_on\ f\ Q;$$

$$q \in Q;$$

$$\delta_states\ \delta \subseteq Q$$

$$\rrbracket \implies accs\ \delta\ n\ q$$

$\langle proof \rangle$

lemma (in $tree_automaton$) $remap_lang2$:

assumes I : $inj_on\ f\ (ta_rstates\ TA)$

shows $t \in ta_lang\ (ta_remap\ f\ TA) \implies t \in ta_lang\ TA$

$\langle proof \rangle$

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

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

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

lemma (in *det-tree-automaton*) *remap-dta*[*intro*, *simp*]:
 assumes *INJ*: $\text{inj-on } f \ Q$
 shows $\text{det-tree-automaton (ta-remap } f \text{ } TA) \ A$
 ⟨proof⟩

lemma (in *complete-tree-automaton*) *remap-cta*[*intro*, *simp*]:
 assumes *INJ*: $\text{inj-on } f \ Q$
 shows $\text{complete-tree-automaton (ta-remap } f \text{ } TA) \ A$
 ⟨proof⟩

3.4.3 Union

definition *ta-union* $TA \ TA' ==$
 ([$\text{ta-initial} = \text{ta-initial } TA \cup \text{ta-initial } TA'$,
 $\text{ta-rules} = \text{ta-rules } TA \cup \text{ta-rules } TA'$
])

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

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

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

lemma *ta-union-correct-aux1*:
fixes $TA TA'$
assumes TA : *tree-automaton* TA
assumes TA' : *tree-automaton* TA'
assumes DJ : *ta-rstates* $TA \cap ta-rstates\ TA' = \{\}$
shows $ta-lang\ (ta-union\ TA\ TA') = ta-lang\ TA \cup ta-lang\ TA'$
 $\langle proof \rangle$

lemma *ta-union-correct-aux2*:
fixes $TA TA'$
assumes TA : *tree-automaton* TA
assumes TA' : *tree-automaton* TA'
shows *tree-automaton* $(ta-union\ TA\ TA')$
 $\langle proof \rangle$

theorem *ta-union-correct*:
fixes $TA TA'$
assumes TA : *tree-automaton* TA
assumes TA' : *tree-automaton* TA'
assumes DJ : *ta-rstates* $TA \cap ta-rstates\ TA' = \{\}$
shows $ta-lang\ (ta-union\ TA\ TA') = ta-lang\ TA \cup ta-lang\ TA'$
tree-automaton $(ta-union\ TA\ TA')$
 $\langle proof \rangle$

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* $(ta-union\ TA\ TA')\ A$
 $\langle proof \rangle$

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

datatype $(q1, q2)\ ustate-wrapper = USW1\ 'q1 \mid USW2\ 'q2$

lemma *usw-disjoint[simp]*:
 $USW1\ 'X \cap USW2\ 'Y = \{\}$
 $remap-rule\ USW1\ 'X \cap remap-rule\ USW2\ 'Y = \{\}$
 $\langle proof \rangle$

lemma *states-usw-disjoint[simp]*:
 $ta-rstates\ (ta-remap\ USW1\ X) \cap ta-rstates\ (ta-remap\ USW2\ Y) = \{\}$
 $\langle proof \rangle$

lemma *usw-inj-on[simp, intro!]*:
 $inj-on\ USW1\ X$
 $inj-on\ USW2\ X$
 $\langle proof \rangle$

definition *ta-union-wrap* $TA\ TA' =$

$ta\text{-union } (ta\text{-remap } USW1 \ TA) \ (ta\text{-remap } USW2 \ TA')$

lemma $ta\text{-union-wrap-correct}$:

fixes $TA :: ('Q1, 'L) \ tree\text{-automaton-rec}$

fixes $TA' :: ('Q2, 'L) \ tree\text{-automaton-rec}$

assumes TA : $tree\text{-automaton } TA$

assumes TA' : $tree\text{-automaton } TA'$

shows $ta\text{-lang } (ta\text{-union-wrap } TA \ TA') = ta\text{-lang } TA \cup ta\text{-lang } TA'$ (**is** $?T1$)
 $tree\text{-automaton } (ta\text{-union-wrap } TA \ TA')$ (**is** $?T2$)

$\langle proof \rangle$

lemma $ta\text{-union-wrap-rta}$:

fixes $TA \ TA'$

assumes TA : $ranked\text{-tree-automaton } TA \ A$

assumes TA' : $ranked\text{-tree-automaton } TA' \ A$

shows $ranked\text{-tree-automaton } (ta\text{-union-wrap } TA \ TA') \ A$

$\langle proof \rangle$

3.4.4 Reduction

definition $reduce\text{-rules } \delta \ P == \delta \cap \{ r. \ rule\text{-states } r \subseteq P \}$

lemma $reduce\text{-rulesI}$: $\llbracket r \in \delta; \ rule\text{-states } r \subseteq P \rrbracket \implies r \in reduce\text{-rules } \delta \ P$

$\langle proof \rangle$

lemma $reduce\text{-rulesD}$:

$\llbracket r \in reduce\text{-rules } \delta \ P \rrbracket \implies r \in \delta$

$\llbracket r \in reduce\text{-rules } \delta \ P; \ q \in rule\text{-states } r \rrbracket \implies q \in P$

$\langle proof \rangle$

lemma $reduce\text{-rules-subset}$: $reduce\text{-rules } \delta \ P \subseteq \delta$

$\langle proof \rangle$

lemma $reduce\text{-rules-mono}$: $P \subseteq P' \implies reduce\text{-rules } \delta \ P \subseteq reduce\text{-rules } \delta \ P'$

$\langle proof \rangle$

lemma $\delta\text{-states-reduce-subset}$:

shows $\delta\text{-states } (reduce\text{-rules } \delta \ Q) \subseteq \delta\text{-states } \delta \cap Q$

$\langle proof \rangle$

lemmas $\delta\text{-states-reduce-subsetI} = rev\text{-subsetD}[OF - \delta\text{-states-reduce-subset}]$

definition $ta\text{-reduce}$

$:: ('Q, 'L) \ tree\text{-automaton-rec} \Rightarrow ('Q \ set) \Rightarrow ('Q, 'L) \ tree\text{-automaton-rec}$

where $ta\text{-reduce } TA \ P ==$

$(\ ta\text{-initial} = ta\text{-initial } TA \cap P,$

$ta\text{-rules} = reduce\text{-rules } (ta\text{-rules } TA) \ P)$

— 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)*
 $\langle proof \rangle$

lemma *reduce- δ -states-rules[simp]*:
 $(ta\text{-rules } (ta\text{-reduce } TA (\delta\text{-states } (ta\text{-rules } TA)))) = ta\text{-rules } TA$
 $\langle proof \rangle$

lemma *ta-reduce- δ -states*:
 $ta\text{-lang } (ta\text{-reduce } TA (\delta\text{-states } (ta\text{-rules } TA))) = ta\text{-lang } TA$
 $\langle proof \rangle$

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

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

inductive-set *f-succ* **for** δ **where**
 $\llbracket (q \rightarrow l \text{ } qs) \in \delta; q' \in \text{set } qs \rrbracket \implies (q, q') \in f\text{-succ } \delta$

— Alternative characterization of forward successors

lemma *f-succ-alt*: $f\text{-succ } \delta = \{(q, q'). \exists l \text{ } qs. (q \rightarrow l \text{ } qs) \in \delta \wedge q' \in \text{set } qs\}$
 $\langle proof \rangle$

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

— 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* $\Rightarrow 'Q$ *set* $\Rightarrow 'Q$ *set*
for δ $Q0$

where

fa-refl: $q0 \in Q0 \implies q0 \in f\text{-accessible-alt } \delta$ $Q0$ |
fa-step: $\llbracket q \in f\text{-accessible-alt } \delta$ $Q0; (q \rightarrow l \text{ } qs) \in \delta; q' \in \text{set } qs \rrbracket$
 $\implies q' \in f\text{-accessible-alt } \delta$ $Q0$

lemma *f-accessible-alt*: $f\text{-accessible } \delta$ $Q0 = f\text{-accessible-alt } \delta$ $Q0$
 $\langle proof \rangle$

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

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

lemma *f-succ-finite[simp, intro]*: *finite* $\delta \implies$ *finite* $(f\text{-succ } \delta)$
 $\langle proof \rangle$

lemma *f-accessible-mono*: $Q \subseteq Q' \implies x \in f\text{-accessible } \delta$ $Q \implies x \in f\text{-accessible } \delta$ Q'
 $\langle proof \rangle$

lemma *f-accessible-prepend*:

[[$(q \rightarrow l qs) \in \delta$; $q' \in \text{set } qs$; $x \in f\text{-accessible } \delta \{q'\}$]]
 $\implies x \in f\text{-accessible } \delta \{q\}$
 <proof>

lemma *f-accessible-subset*: $q \in f\text{-accessible } \delta Q \implies q \in Q \cup \delta\text{-states } \delta$
 <proof>

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

lemma *f-accessible-refl-inter-simp*[*simp*]: $Q \cap f\text{-accessible } r Q = Q$
 <proof>

lemma *accs-reduce-f-acc*:
 $accs \delta t q \implies accs (reduce\text{-rules } \delta (f\text{-accessible } \delta \{q\})) t q$
 <proof>

abbreviation *ta-fwd-reduce* $TA ==$
 $(ta\text{-reduce } TA (f\text{-accessible } (ta\text{-rules } TA) (ta\text{-initial } TA)))$

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

theorem *ta-reduce-f-acc*[*simp*]: $ta\text{-lang } (ta\text{-fwd-reduce } TA) = ta\text{-lang } TA$
 <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* :: $('Q, 'L) ta\text{-rule set} \Rightarrow 'Q \text{ set}$
 for δ
 where
 [[$(q \rightarrow l qs) \in \delta$; $\forall x. x \in \text{set } qs \implies x \in b\text{-accessible } \delta$]] $\implies q \in b\text{-accessible } \delta$

lemma *b-accessibleI*:
 [[$(q \rightarrow l qs) \in \delta$; $\text{set } qs \subseteq b\text{-accessible } \delta$]] $\implies q \in b\text{-accessible } \delta$
 <proof>

lemma *accs-is-b-accessible*: $accs \delta t q \implies q \in b\text{-accessible } \delta$
 <proof>

lemma *b-acc-subset- δ -statesI*: $x \in b\text{-accessible } \delta \implies x \in \delta\text{-states } \delta$
 <proof>

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

lemma *b-acc-finite*[*simp, intro!*]: $\text{finite } \delta \implies \text{finite } (b\text{-accessible } \delta)$
 <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* $\delta 1$ $\delta 2$ == {

r-prod (*q1* → *l* *qs1*) (*q2* → *l* *qs2*) | *q1* *q2* *l* *qs1* *qs2*.
length *qs1* = *length* *qs2* ∧
 (*q1* → *l* *qs1*) ∈ $\delta 1$ ∧
 (*q2* → *l* *qs2*) ∈ $\delta 2$

}

lemma δ -*prodI*: [[

length *qs1* = *length* *qs2*;
 (*q1* → *l* *qs1*) ∈ $\delta 1$;
 (*q2* → *l* *qs2*) ∈ $\delta 2$]] ⇒ ((*q1*, *q2*) → *l* (*zip* *qs1* *qs2*)) ∈ δ -*prod* $\delta 1$ $\delta 2$

⟨*proof*⟩

lemma δ -*prodE*:

[[
r ∈ δ -*prod* $\delta 1$ $\delta 2$;
 !!*q1* *q2* *l* *qs1* *qs2*. [[*length* *qs1* = *length* *qs2*;
 (*q1* → *l* *qs1*) ∈ $\delta 1$;
 (*q2* → *l* *qs2*) ∈ $\delta 2$;
r = ((*q1*, *q2*) → *l* (*zip* *qs1* *qs2*))
]] ⇒ *P*

]] ⇒ *P*

⟨*proof*⟩

lemma δ -*prod-sound*:

assumes $A: \text{accs } (\delta\text{-prod } \delta 1 \ \delta 2) \ t \ (q1, q2)$
shows $\text{accs } \delta 1 \ t \ q1 \quad \text{accs } \delta 2 \ t \ q2$
 $\langle \text{proof} \rangle$
lemma $\delta\text{-prod-precise}$:
 $\llbracket \text{accs } \delta 1 \ t \ q1; \text{accs } \delta 2 \ t \ q2 \rrbracket \implies \text{accs } (\delta\text{-prod } \delta 1 \ \delta 2) \ t \ (q1, q2)$
 $\langle \text{proof} \rangle$

lemma $\delta\text{-prod-empty}[simp]$:
 $\delta\text{-prod } \{\} \ \delta = \{\}$
 $\delta\text{-prod } \delta \ \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma $\delta\text{-prod-2sng}[simp]$:
 $\llbracket \text{rhsl } r1 \neq \text{rhsl } r2 \rrbracket \implies \delta\text{-prod } \{r1\} \ \{r2\} = \{\}$
 $\llbracket \text{length } (\text{rhsq } r1) \neq \text{length } (\text{rhsq } r2) \rrbracket \implies \delta\text{-prod } \{r1\} \ \{r2\} = \{\}$
 $\llbracket \text{rhsl } r1 = \text{rhsl } r2; \text{length } (\text{rhsq } r1) = \text{length } (\text{rhsq } r2) \rrbracket$
 $\implies \delta\text{-prod } \{r1\} \ \{r2\} = \{r\text{-prod } r1 \ r2\}$
 $\langle \text{proof} \rangle$

lemma $\delta\text{-prod-Un}[simp]$:
 $\delta\text{-prod } (\delta 1 \cup \delta 1') \ \delta 2 = \delta\text{-prod } \delta 1 \ \delta 2 \cup \delta\text{-prod } \delta 1' \ \delta 2$
 $\delta\text{-prod } \delta 1 \ (\delta 2 \cup \delta 2') = \delta\text{-prod } \delta 1 \ \delta 2 \cup \delta\text{-prod } \delta 1 \ \delta 2'$
 $\langle \text{proof} \rangle$

The next two definitions are solely for technical reasons. They are required to allow simplification of expressions of the form $\delta\text{-prod } (\text{insert } r \ \delta 1) \ \delta 2$ or $\delta\text{-prod } \delta 1 \ (\text{insert } r \ \delta 2)$, without making the simplifier loop.

definition $\delta\text{-prod-sng1 } r \ \delta 2 ==$
 $\text{case } r \text{ of } (q1 \rightarrow l \ qs1) \Rightarrow$
 $\{ r\text{-prod } r \ (q2 \rightarrow l \ qs2) \mid$
 $q2 \ qs2. \text{length } qs1 = \text{length } qs2 \wedge (q2 \rightarrow l \ qs2) \in \delta 2$
 $\}$

definition $\delta\text{-prod-sng2 } \delta 1 \ r ==$
 $\text{case } r \text{ of } (q2 \rightarrow l \ qs2) \Rightarrow$
 $\{ r\text{-prod } (q1 \rightarrow l \ qs1) \ r \mid$
 $q1 \ qs1. \text{length } qs1 = \text{length } qs2 \wedge (q1 \rightarrow l \ qs1) \in \delta 1$
 $\}$

lemma $\delta\text{-prod-sng-alt}$:
 $\delta\text{-prod-sng1 } r \ \delta 2 = \delta\text{-prod } \{r\} \ \delta 2$
 $\delta\text{-prod-sng2 } \delta 1 \ r = \delta\text{-prod } \delta 1 \ \{r\}$
 $\langle \text{proof} \rangle$

lemmas $\delta\text{-prod-insert} =$
 $\delta\text{-prod-Un}(1)[\text{where } ?\delta 1.0 = \{x\}, \text{ simplified, folded } \delta\text{-prod-sng-alt}]$
 $\delta\text{-prod-Un}(2)[\text{where } ?\delta 2.0 = \{x\}, \text{ simplified, folded } \delta\text{-prod-sng-alt}]$
for x

— Product automaton

definition $ta\text{-prod } TA1 \ TA2 ==$
 \langle $ta\text{-initial} = ta\text{-initial } TA1 \times ta\text{-initial } TA2,$
 $ta\text{-rules} = \delta\text{-prod } (ta\text{-rules } TA1) (ta\text{-rules } TA2)$
 \rangle

lemma $ta\text{-prod-correct-aux1}$:
 $ta\text{-lang } (ta\text{-prod } TA1 \ TA2) = ta\text{-lang } TA1 \cap ta\text{-lang } TA2$
 $\langle proof \rangle$

lemma $\delta\text{-states-cart}$:
 $q \in \delta\text{-states } (\delta\text{-prod } \delta1 \ \delta2) \implies q \in \delta\text{-states } \delta1 \times \delta\text{-states } \delta2$
 $\langle proof \rangle$

lemma $\delta\text{-prod-finite}$ [*simp, intro*]:
 $finite \ \delta1 \implies finite \ \delta2 \implies finite \ (\delta\text{-prod } \delta1 \ \delta2)$
 $\langle proof \rangle$

lemma $ta\text{-prod-correct-aux2}$:
assumes TA : $tree\text{-automaton } TA1 \ tree\text{-automaton } TA2$
shows $tree\text{-automaton } (ta\text{-prod } TA1 \ TA2)$
 $\langle proof \rangle$

theorem $ta\text{-prod-correct}$:
assumes TA : $tree\text{-automaton } TA1 \ tree\text{-automaton } TA2$
shows
 $ta\text{-lang } (ta\text{-prod } TA1 \ TA2) = ta\text{-lang } TA1 \cap ta\text{-lang } TA2$
 $tree\text{-automaton } (ta\text{-prod } TA1 \ TA2)$
 $\langle proof \rangle$

lemma $ta\text{-prod-rta}$:
assumes TA : $ranked\text{-tree-automaton } TA1 \ A \ ranked\text{-tree-automaton } TA2 \ A$
shows $ranked\text{-tree-automaton } (ta\text{-prod } TA1 \ TA2) \ A$
 $\langle proof \rangle$

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 $ranked\text{-tree-automaton}$

begin

— Left-hand side of subset rule for given symbol and rhs

definition $\delta ss\text{-lhs } f \ ss ==$
 $\{ q \mid q \text{ qs. } (q \rightarrow f \ qs) \in \delta \wedge list\text{-all-zip } (\in) \ qs \ ss \}$

— Subset construction

inductive-set $\delta ss :: ('Q \ set, 'L) \ ta\text{-rule } set \ \mathbf{where}$

$\llbracket A f = \text{Some } (\text{length } ss);$
 $ss \in \text{lists } \{s. s \subseteq \text{ta-rstates } TA\};$
 $s = \delta ss\text{-lhs } f ss$
 $\rrbracket \implies (s \rightarrow f ss) \in \delta ss$

lemma δssI :

assumes $A: A f = \text{Some } (\text{length } ss)$
 $ss \in \text{lists } \{s. s \subseteq \text{ta-rstates } TA\}$

shows

$(\delta ss\text{-lhs } f ss) \rightarrow f ss) \in \delta ss$
 $\langle \text{proof} \rangle$

lemma $\delta ss\text{-subset}[\text{simp}, \text{intro!}]$: $\delta ss\text{-lhs } f ss \subseteq Q$
 $\langle \text{proof} \rangle$

lemma $\delta ss\text{-finite}[\text{simp}, \text{intro!}]$: $\text{finite } \delta ss$
 $\langle \text{proof} \rangle$

lemma $\delta ss\text{-det}$: $\llbracket (q \rightarrow f qs) \in \delta ss; (q' \rightarrow f qs) \in \delta ss \rrbracket \implies q = q'$
 $\langle \text{proof} \rangle$

lemma $\delta ss\text{-accs-sound}$:

assumes $A: \text{accs } \delta t q$

obtains s **where**

$s \subseteq Q$

$q \in s$

$\text{accs } \delta ss t s$

$\langle \text{proof} \rangle$

lemma $\delta ss\text{-accs-precise}$:

assumes $A: \text{accs } \delta ss t s \quad q \in s$

shows $\text{accs } \delta t q$

$\langle \text{proof} \rangle$

definition $\text{detTA} == (\mid \text{ta-initial} = \{s. s \subseteq Q \wedge s \cap Qi \neq \{\}\},$
 $\text{ta-rules} = \delta ss \mid)$

theorem $\text{detTA-is-ta}[\text{simp}, \text{intro}]$:

$\text{det-tree-automaton } \text{detTA } A$

$\langle \text{proof} \rangle$

theorem $\text{detTA-lang}[\text{simp}]$:

$\text{ta-lang } (\text{detTA}) = \text{ta-lang } TA$

$\langle \text{proof} \rangle$

lemmas $\text{detTA-correct} = \text{detTA-is-ta } \text{detTA-lang}$
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 $Qcomplete\text{-}finite[simp, intro!]:\ finite\ Qcomplete$

<proof>

definition $\delta complete :: ('Q\ option, 'L)\ ta\text{-}rule\ set\ \mathbf{where}$

$\delta complete == (remap\text{-}rule\ Some\ ' \delta)$

$\cup \{ (None \rightarrow f\ qs) \mid f\ qs.$

$A\ f = Some\ (length\ qs)$

$\wedge\ qs \in lists\ Qcomplete$

$\wedge\ \neg(\exists\ qo\ qso.\ (qo \rightarrow f\ qso) \in \delta \wedge qs = map\ Some\ qso) \}$

lemma $\delta\text{-}states\text{-}complete: q \in \delta\text{-}states\ \delta complete \implies q \in Qcomplete$

<proof>

definition

$completeTA == (\mid ta\text{-}initial = Some\ 'Qi,\ ta\text{-}rules = \delta complete \mid)$

lemma $\delta complete\text{-}finite[simp, intro]:\ finite\ \delta complete$

<proof>

theorem $completeTA\text{-}is\text{-}ta: complete\text{-}tree\text{-}automaton\ completeTA\ A$

<proof>

theorem $completeTA\text{-}lang: ta\text{-}lang\ completeTA = ta\text{-}lang\ TA$

<proof>

lemmas $completeTA\text{-}correct = completeTA\text{-}is\text{-}ta\ completeTA\text{-}lang$

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* == \langle
ta-initial = $Q - Qi$,
ta-rules = δ \rangle

lemma *cta-rules[simp]*: *ta-rules complementTA* = δ
 \langle *proof* \rangle

theorem *complementTA-correct*:
ta-lang complementTA = *ranked-trees A - ta-lang TA* (**is** ?T1)
complete-tree-automaton complementTA A (**is** ?T2)
 \langle *proof* \rangle

end

3.5 Regular Tree Languages

3.5.1 Definitions

definition *regular-languages* :: $('L \rightarrow \text{nat}) \Rightarrow 'L \text{ tree set set}$
where *regular-languages A* ==
 $\{ \text{ta-lang TA} \mid (\text{TA}::(\text{nat}, 'L) \text{ tree-automaton-rec}).$
 $\text{ranked-tree-automaton TA A} \}$

lemma *rtlE*:
fixes $L :: 'L \text{ tree set}$
assumes $A: L \in \text{regular-languages A}$
obtains $\text{TA}::(\text{nat}, 'L) \text{ tree-automaton-rec}$ **where**
 $L = \text{ta-lang TA}$
 $\text{ranked-tree-automaton TA A}$
 \langle *proof* \rangle

context *ranked-tree-automaton*
begin

lemma (**in** *ranked-tree-automaton*) *rtlI[simp]*:
shows $\text{ta-lang TA} \in \text{regular-languages A}$
 \langle *proof* \rangle

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

theorem *obtain-complete*:
obtains $\text{TAC}::('Q \text{ set option}, 'L) \text{ tree-automaton-rec}$ **where**
 $\text{ta-lang TAC} = \text{ta-lang TA}$
 $\text{complete-tree-automaton TAC A}$
 \langle *proof* \rangle

end

lemma *rtlE-complete*:
fixes $L :: 'L$ tree set
assumes $A: L \in \text{regular-languages } A$
obtains $TA :: (\text{nat}, 'L)$ tree-automaton-rec **where**
 $L = \text{ta-lang } TA$
 $\text{complete-tree-automaton } TA \ A$
 $\langle \text{proof} \rangle$

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!]*: $\{\} \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

theorem *rtl-union-closed*:
 $\llbracket L1 \in \text{regular-languages } A; L2 \in \text{regular-languages } A \rrbracket$
 $\implies L1 \cup L2 \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

theorem *rtl-inter-closed*:
 $\llbracket L1 \in \text{regular-languages } A; L2 \in \text{regular-languages } A \rrbracket \implies$
 $L1 \cap L2 \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

theorem *rtl-complement-closed*:
 $L \in \text{regular-languages } A \implies \text{ranked-trees } A - L \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

theorem (in *finite-alphabet*) *rtl-univ*:
 $\text{ranked-trees } A \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

theorem *rtl-diff-closed*:
fixes $L1 :: 'L$ tree set
assumes $A[\text{simp}]$: $L1 \in \text{regular-languages } A \quad L2 \in \text{regular-languages } A$
shows $L1 - L2 \in \text{regular-languages } A$
 $\langle \text{proof} \rangle$

lemmas *rtl-closed = finite-alphabet.rtl-empty finite-alphabet.rtl-univ*
 $\text{rtl-complement-closed}$
 $\text{rtl-inter-closed rtl-union-closed rtl-diff-closed}$

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
```

— *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. !!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 (λt. { q . accs δ t q }) ts) (is ?T2)
  <proof>
```

theorem *faccs-correct1*: $q \in \text{faccs } \delta \ n \implies \text{accs } \delta \ n \ q$
 ⟨*proof*⟩

theorem *faccs-correct2*: $\text{accs } \delta \ n \ q \implies q \in \text{faccs } \delta \ n$
 ⟨*proof*⟩

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

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

4.2 Backward Reduction and Emptiness Check

4.2.1 Auxiliary Definitions

inductive-set *bacc-step* :: $('Q, 'L) \text{ ta-rule set} \Rightarrow 'Q \text{ set} \Rightarrow 'Q \text{ set}$
 for $\delta \ Q$

where

$\llbracket r \in \delta; \text{set } (r \text{hsq } r) \subseteq Q \rrbracket \implies \text{lhs } r \in \text{bacc-step } \delta \ Q$

— 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*: $\text{bacc-step } \delta \ Q \subseteq Q$

shows *b-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** $('Q, 'L) \text{ br-state} = 'Q \text{ set} \times 'Q \text{ set}$

- Set of states that are non-empty (accept a tree) after adding the state q to the set of discovered states

definition *br-dsq*

$:: ('Q, 'L) \text{ ta-rule set} \Rightarrow 'Q \Rightarrow ('Q, 'L) \text{ br-state} \Rightarrow 'Q \text{ set}$

where

$\text{br-dsq } \delta \ q == \lambda(Q, W). \{ \text{lhs } r \mid r. r \in \delta \wedge \text{set } (\text{rhs } r) \subseteq (Q - (W - \{q\})) \}$

- 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*

$:: ('Q, 'L) \text{ ta-rule set} \Rightarrow (('Q, 'L) \text{ br-state} \times ('Q, 'L) \text{ br-state}) \text{ set}$

for δ **where**

\llbracket
 $q \in W;$
 $Q' = Q \cup \text{br-dsq } \delta \ q \ (Q, W);$
 $W' = W - \{q\} \cup (\text{br-dsq } \delta \ q \ (Q, W) - Q)$
 $\rrbracket \Rightarrow ((Q, W), (Q', W')) \in \text{br-step } \delta$

- Termination condition for backwards reduction: The workset is empty

definition *br-cond* $:: ('Q, 'L) \text{ br-state set}$

where $\text{br-cond} == \{(Q, W). W \neq \{\}\}$

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

definition *bre-cond* $:: 'Q \text{ set} \Rightarrow ('Q, 'L) \text{ br-state set}$

where $\text{bre-cond } Qi == \{(Q, W). W \neq \{\} \wedge (Qi \cap Q = \{\})\}$

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

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

where $\text{br-iq } \delta == \{ \text{lhs } r \mid r. r \in \delta \wedge \text{rhs } r = [] \}$

- Initial state for the iteration

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

where $\text{br-initial } \delta == (\text{br-iq } \delta, \text{br-iq } \delta)$

- 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* $:: ('Q, 'L) \text{ ta-rule set} \Rightarrow ('Q, 'L) \text{ br-state set}$

where $\text{br-invar } \delta == \{(Q, W).$

$W \subseteq Q \wedge$

$$\begin{aligned} & Q \subseteq b\text{-accessible } \delta \wedge \\ & b\text{acc-step } \delta (Q - W) \subseteq Q \wedge \\ & \text{finite } Q \} \end{aligned}$$

definition $br\text{-algo } \delta == \langle$
 $wa\text{-cond} = br\text{-cond},$
 $wa\text{-step} = br\text{-step } \delta,$
 $wa\text{-initial} = \{br\text{-initial } \delta\},$
 $wa\text{-invar} = br\text{-invar } \delta$
 \rangle

definition $bre\text{-algo } Qi \delta == \langle$
 $wa\text{-cond} = bre\text{-cond } Qi,$
 $wa\text{-step} = br\text{-step } \delta,$
 $wa\text{-initial} = \{br\text{-initial } \delta\},$
 $wa\text{-invar} = br\text{-invar } \delta$
 \rangle

— Termination: Either a new state is added, or the workset decreases.

definition $br\text{-termrel } \delta ==$
 $\langle \{(Q', Q). Q \subset Q' \wedge Q' \subseteq b\text{-accessible } \delta\} \langle *lex* \rangle \text{finite-psubset}$

lemma $bre\text{-cond-imp-br-cond}[\text{intro}, \text{simp}]: bre\text{-cond } Qi \subseteq br\text{-cond}$
 $\langle \text{proof} \rangle$

lemma $br\text{-termrel-wf}[\text{simp}, \text{intro!}]: \text{finite } \delta \implies wf (br\text{-termrel } \delta)$
 $\langle \text{proof} \rangle$

lemma $br\text{-dsq-ss}:$
assumes $A: (Q, W) \in br\text{-invar } \delta \quad W \neq \{\} \quad q \in W$
shows $br\text{-dsq } \delta q (Q, W) \subseteq b\text{-accessible } \delta$
 $\langle \text{proof} \rangle$

lemma $br\text{-step-in-termrel}:$
assumes $A: \Sigma \in br\text{-cond} \quad \Sigma \in br\text{-invar } \delta \quad (\Sigma, \Sigma') \in br\text{-step } \delta$
shows $(\Sigma', \Sigma) \in br\text{-termrel } \delta$
 $\langle \text{proof} \rangle$

lemma $br\text{-invar-initial}[\text{simp}]: \text{finite } \delta \implies (br\text{-initial } \delta) \in br\text{-invar } \delta$
 $\langle \text{proof} \rangle$

lemma $br\text{-invar-step}:$
assumes $[\text{simp}]: \text{finite } \delta$
assumes $A: \Sigma \in br\text{-cond} \quad \Sigma \in br\text{-invar } \delta \quad (\Sigma, \Sigma') \in br\text{-step } \delta$
shows $\Sigma' \in br\text{-invar } \delta$
 $\langle \text{proof} \rangle$

lemma $br\text{-invar-final}:$

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

theorem *br-while-algo*:
assumes $FIN[simp]$: finite δ
shows *while-algo* ($br\text{-algo } \delta$)
 ⟨proof⟩

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

theorem *bre-while-algo*:
assumes $FIN[simp]$: finite δ
shows *while-algo* ($bre\text{-algo } Qi \ \delta$)
 ⟨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 $('Q, 'L) \text{ br}'\text{-state} = 'Q \text{ set} \times 'Q \text{ set} \times (('Q, 'L) \text{ ta-rule} \rightarrow \text{nat})$

— Abstraction to α -level

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

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

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

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

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

definition $br'\text{-cond} :: ('Q, 'L) br'\text{-state set}$

where $br'\text{-cond} == \{(Q, W, rcm). W \neq \{\}\}$

definition $bre'\text{-cond} :: 'Q\ set \Rightarrow ('Q, 'L) br'\text{-state set}$

where $bre'\text{-cond } Qi == \{(Q, W, rcm). W \neq \{\} \wedge (Qi \cap Q = \{\})\}$

inductive-set $br'\text{-initial} :: ('Q, 'L) ta\text{-rule set} \Rightarrow ('Q, 'L) br'\text{-state set}$

for δ **where**

$$\begin{aligned}
\llbracket &!!r. r \in \delta \implies rcm\ r = \text{Some } (card\ (set\ (rhsq\ r))) \rrbracket \\
&\implies (br\text{-iq } \delta, br\text{-iq } \delta, rcm) \in br'\text{-initial } \delta
\end{aligned}$$

definition $br'\text{-algo } \delta == ($

$$\begin{aligned}
&wa\text{-cond} = br'\text{-cond}, \\
&wa\text{-step} = br'\text{-step } \delta, \\
&wa\text{-initial} = br'\text{-initial } \delta, \\
&wa\text{-invar} = br'\text{-invar } \delta
\end{aligned}$$

)

definition $bre'\text{-algo } Qi\ \delta == ($

$$\begin{aligned}
&wa\text{-cond} = bre'\text{-cond } Qi, \\
&wa\text{-step} = br'\text{-step } \delta, \\
&wa\text{-initial} = br'\text{-initial } \delta, \\
&wa\text{-invar} = br'\text{-invar } \delta
\end{aligned}$$

)

lemma $br'\text{-step-invar}$:

assumes $finite[simp]$: $finite\ \delta$

assumes INV : $\Sigma \in br'\text{-invar-add } \delta \quad br'\text{-}\alpha\ \Sigma \in br\text{-invar } \delta$

assumes $STEP$: $(\Sigma, \Sigma') \in br'\text{-step } \delta$

shows $\Sigma' \in br'\text{-invar-add } \delta$

$\langle proof \rangle$

lemma $br'\text{-invar-initial}$:

$br'\text{-initial } \delta \subseteq br'\text{-invar-add } \delta$

$\langle proof \rangle$

lemma $br'\text{-rcm-aux}'$:

$\llbracket (Q, W, rcm) \in br'\text{-invar } \delta; q \in W \rrbracket$

$\implies \{r \in \delta. q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$

$= \{r \in \delta. q \in set\ (rhsq\ r) \wedge set\ (rhsq\ r) \subseteq (Q - (W - \{q\}))\}$

$\langle proof \rangle$

lemma *br'-rcm-aux*:

assumes $A: (Q, W, rcm) \in br'\text{-invar } \delta \quad q \in W$

shows $\{lhs\ r \mid r. r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$

$= \{lhs\ r \mid r. r \in \delta \wedge q \in set\ (rhsq\ r) \wedge set\ (rhsq\ r) \subseteq (Q - (W - \{q\}))\}$

$\langle proof \rangle$

lemma *br'-invar-QcD*:

$(Q, W, rcm) \in br'\text{-invar } \delta \implies \{lhs\ r \mid r. r \in \delta \wedge set\ (rhsq\ r) \subseteq (Q - W)\} \subseteq Q$

$\langle proof \rangle$

lemma *br'-rcm-aux2*:

$\llbracket (Q, W, rcm) \in br'\text{-invar } \delta; q \in W \rrbracket$

$\implies Q \cup br\text{-dsq } \delta\ q\ (Q, W)$

$= Q \cup \{lhs\ r \mid r. r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\}$

$\langle proof \rangle$

lemma *br'-rcm-aux3*:

$\llbracket (Q, W, rcm) \in br'\text{-invar } \delta; q \in W \rrbracket$

$\implies br\text{-dsq } \delta\ q\ (Q, W) - Q$

$= \{lhs\ r \mid r. r \in \delta \wedge q \in set\ (rhsq\ r) \wedge the\ (rcm\ r) \leq Suc\ 0\} - Q$

$\langle proof \rangle$

lemma *br'-step-abs*:

\llbracket

$\Sigma \in br'\text{-invar } \delta;$

$(\Sigma, \Sigma') \in br'\text{-step } \delta$

$\rrbracket \implies (br'\text{-}\alpha\ \Sigma, br'\text{-}\alpha\ \Sigma') \in br\text{-step } \delta$

$\langle proof \rangle$

lemma *br'-initial-abs*: $br'\text{-}\alpha\ (br'\text{-initial } \delta) = \{br\text{-initial } \delta\}$

$\langle proof \rangle$

lemma *br'-cond-abs*: $\Sigma \in br'\text{-cond} \iff (br'\text{-}\alpha\ \Sigma) \in br\text{-cond}$

$\langle proof \rangle$

lemma *bre'-cond-abs*: $\Sigma \in bre'\text{-cond } Qi \iff (br'\text{-}\alpha\ \Sigma) \in bre\text{-cond } Qi$

$\langle proof \rangle$

lemma *br'-invar-abs*: $br'\text{-}\alpha\ (br'\text{-invar } \delta) \subseteq br\text{-invar } \delta$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

theorem *br'-while-algo*:

finite $\delta \implies \text{while-algo } (br'\text{-algo } \delta)$
<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 *br'-pref-br*: *wa-precise-refine* (*bre'-algo Qi* δ) (*bre-algo Qi* δ) *br'- α*

<proof>

interpretation *br'-pref*:

wa-precise-refine br'-algo Qi δ *bre-algo Qi* δ *br'- α*
<proof>

theorem *bre'-while-algo*:

finite $\delta \implies \text{while-algo } (bre'\text{-algo } Qi \delta)$
<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* \Rightarrow (*'Q, 'L*) *br'-state* \Rightarrow (*'Q, 'L*) *br'-state*

where

br'-inner-step == $\lambda r (Q, W, rcm). \text{let } c = \text{the } (rcm \ r) \text{ in } ($
 if $c \leq 1$ *then insert* (*lhs r*) *Q* *else Q,*
 if $c \leq 1 \wedge (\text{lhs } r) \notin Q$ *then insert* (*lhs r*) *W* *else W,*
 rcm ($r \mapsto (c - (1 :: \text{nat}))$)
 $)$

definition *br'-inner-invar*

:: (*'Q, 'L*) *ta-rule set* \Rightarrow *'Q* \Rightarrow (*'Q, 'L*) *br'-state*
 \Rightarrow (*'Q, 'L*) *ta-rule set* \Rightarrow (*'Q, 'L*) *br'-state* \Rightarrow *bool*

where

br'-inner-invar rules q == $\lambda(Q, W, rcm) \text{ it } (Q', W', rcm')$.
 $Q' = Q \cup \{ \text{lhs } r \mid r. r \in \text{rules-it} \wedge \text{the } (rcm \ r) \leq 1 \} \wedge$
 $W' = (W - \{q\}) \cup (\{ \text{lhs } r \mid r. r \in \text{rules-it} \wedge \text{the } (rcm \ r) \leq 1 \} - Q) \wedge$
 $(\forall r. rcm' \ r = (\text{if } r \in \text{rules-it} \text{ then } \text{Some } (\text{the } (rcm \ r) - 1) \text{ else } rcm \ r))$

lemma *br'-inner-invar-imp-final*:

$$\llbracket q \in W; br'\text{-inner-invar } \{r \in \delta. q \in \text{set } (rhsq\ r)\} q (Q, W - \{q\}, rcm) \{\} \Sigma' \rrbracket$$

$$\implies ((Q, W, rcm), \Sigma') \in br'\text{-step } \delta$$
 <proof>

lemma *br'-inner-invar-step*:

$$\llbracket q \in W; br'\text{-inner-invar } \{r \in \delta. q \in \text{set } (rhsq\ r)\} q (Q, W - \{q\}, rcm) \textit{it } \Sigma';$$

$$r \in \textit{it}; \textit{it} \subseteq \{r \in \delta. q \in \text{set } (rhsq\ r)\}$$

$$\rrbracket \implies br'\text{-inner-invar } \{r \in \delta. q \in \text{set } (rhsq\ r)\} q (Q, W - \{q\}, rcm)$$

$$(\textit{it} - \{r\}) (br'\text{-inner-step } r\ \Sigma')$$

<proof>

lemma *br'-inner-invar-initial*:

$$\llbracket q \in W \rrbracket \implies br'\text{-inner-invar } \{r \in \delta. q \in \text{set } (rhsq\ r)\} q (Q, W - \{q\}, rcm)$$

$$\{r \in \delta. q \in \text{set } (rhsq\ r)\} (Q, W - \{q\}, rcm)$$

<proof>

lemma *br'-inner-step-proof*:

fixes $\alpha s :: ' \Sigma \Rightarrow ('Q, 'L) \textit{br}'\text{-state}$
fixes $cstep :: ('Q, 'L) \textit{ta-rule} \Rightarrow ' \Sigma \Rightarrow ' \Sigma$
fixes $\Sigma h :: ' \Sigma$
fixes $cinvar :: ('Q, 'L) \textit{ta-rule set} \Rightarrow ' \Sigma \Rightarrow \textit{bool}$

assumes *iterable-set*: $\textit{set-iteratei } \alpha \textit{invar iteratei}$

assumes *invar-initial*: $cinvar \{r \in \delta. q \in \text{set } (rhsq\ r)\} \Sigma h$

assumes *invar-step*:

$$\llbracket \textit{it } r\ \Sigma. \llbracket r \in \textit{it}; \textit{it} \subseteq \{r \in \delta. q \in \text{set } (rhsq\ r)\}; cinvar \textit{it } \Sigma \rrbracket$$

$$\implies cinvar (\textit{it} - \{r\}) (cstep\ r\ \Sigma)$$

assumes *step-desc*:

$$\llbracket \textit{it } r\ \Sigma. \llbracket r \in \textit{it}; \textit{it} \subseteq \{r \in \delta. q \in \text{set } (rhsq\ r)\}; cinvar \textit{it } \Sigma \rrbracket$$

$$\implies \alpha s (cstep\ r\ \Sigma) = br'\text{-inner-step } r\ (\alpha s\ \Sigma)$$

assumes *it-set-desc*: $\textit{invar it-set} \quad \alpha \textit{it-set} = \{r \in \delta. q \in \text{set } (rhsq\ r)\}$

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 $(\alpha s\ \Sigma, \alpha s (\textit{iteratei it-set } (\lambda-. \textit{True})\ cstep\ \Sigma h)) \in br'\text{-step } \delta$
 <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* $\delta\ m == \forall q\ t. m\ q = \textit{Some } t \longrightarrow \textit{accs } \delta\ t\ q$

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

definition *construct-witness*

$:: ('Q \rightarrow 'L \text{ tree}) \Rightarrow ('Q, 'L) \text{ ta-rule} \Rightarrow 'L \text{ tree}$

where

$\text{construct-witness } Q \ r == \text{NODE } (\text{rhsl } r) (\text{List.map } (\lambda q. \text{the } (Q \ q)) (\text{rhsq } r))$

lemma *witness-propD*: $\llbracket \text{witness-prop } \delta \ m; \ m \ q = \text{Some } t \rrbracket \Longrightarrow \text{accs } \delta \ t \ q$
 $\langle \text{proof} \rangle$

lemma *construct-witness-correct*:

$\llbracket \text{witness-prop } \delta \ Q; \ r \in \delta; \ \text{set } (\text{rhsq } r) \subseteq \text{dom } Q \rrbracket$
 $\Longrightarrow \text{accs } \delta \ (\text{construct-witness } Q \ r) \ (\text{lhs } r)$
 $\langle \text{proof} \rangle$

lemma *construct-witness-eq*:

$\llbracket Q \mid \text{set } (\text{rhsq } r) = Q' \mid \text{set } (\text{rhsq } r) \rrbracket \Longrightarrow$
 $\text{construct-witness } Q \ r = \text{construct-witness } Q' \ r$
 $\langle \text{proof} \rangle$

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

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

definition *brw- α* $:: ('Q, 'L) \text{ brw-state} \Rightarrow ('Q, 'L) \text{ br}'\text{-state}$

where $\text{brw-}\alpha = (\lambda(Q, W, \text{rcm}). (\text{dom } Q, W, \text{rcm}))$

definition *brw-invar-add* $:: ('Q, 'L) \text{ ta-rule set} \Rightarrow ('Q, 'L) \text{ brw-state set}$

where $\text{brw-invar-add } \delta == \{(Q, W, \text{rcm}). \text{witness-prop } \delta \ Q\}$

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

inductive-set *brw-step*

$:: ('Q, 'L) \text{ ta-rule set} \Rightarrow (('Q, 'L) \text{ brw-state} \times ('Q, 'L) \text{ brw-state}) \text{ set}$

for δ **where**

\llbracket
 $q \in W;$
 $\text{dsqr} = \{ r \in \delta. q \in \text{set } (\text{rhsq } r) \wedge \text{the } (\text{rcm } r) \leq 1 \};$
 $\text{dom } Q' = \text{dom } Q \cup \text{lhs}'\text{dsqr};$
 $!!q \ t. \ Q' \ q = \text{Some } t \Longrightarrow Q \ q = \text{Some } t$
 $\quad \vee (\exists r \in \text{dsqr}. q = \text{lhs } r \wedge t = \text{construct-witness } Q \ r);$
 $W' = (W - \{q\}) \cup (\text{lhs}'\text{dsqr} - \text{dom } Q);$
 $!!r. \ r \in \delta \Longrightarrow \text{rcm}' \ r = (\text{if } q \in \text{set } (\text{rhsq } r) \text{ then}$
 $\quad \text{Some } (\text{the } (\text{rcm } r) - 1)$
 $\quad \text{else } \text{rcm } r$
 $\quad)$
 $\rrbracket \Longrightarrow ((Q, W, \text{rcm}), (Q', W', \text{rcm}')) \in \text{brw-step } \delta$

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

inductive-set $brw\text{-}iq :: ('Q, 'L) \text{ ta}\text{-}rule \ \text{set} \Rightarrow ('Q \rightarrow 'L \ \text{tree}) \ \text{set}$
for δ **where**

\llbracket
 $\forall q \ t. \ Q \ q = \text{Some } t \longrightarrow (\exists r \in \delta. \ \text{rhsq } r = [] \wedge q = \text{lhs } r$
 $\qquad \qquad \qquad \wedge t = \text{NODE } (\text{rhsl } r) \ \llbracket \rrbracket);$
 $\forall r \in \delta. \ \text{rhsq } r = [] \longrightarrow Q \ (\text{lhs } r) \neq \text{None}$
 $\rrbracket \Longrightarrow Q \in brw\text{-}iq \ \delta$

inductive-set $brw\text{-}initial :: ('Q, 'L) \text{ ta}\text{-}rule \ \text{set} \Rightarrow ('Q, 'L) \text{ brw}\text{-}state \ \text{set}$
for δ **where**

$\llbracket \forall r. \ r \in \delta \Longrightarrow rcm \ r = \text{Some } (\text{card } (\text{set } (\text{rhsq } r))); \ Q \in brw\text{-}iq \ \delta \rrbracket$
 $\Longrightarrow (Q, br\text{-}iq \ \delta, rcm) \in brw\text{-}initial \ \delta$

definition $brw\text{-}algo \ Qi \ \delta == \langle$
 $\ \ wa\text{-}cond = brw\text{-}cond \ Qi,$
 $\ \ wa\text{-}step = brw\text{-}step \ \delta,$
 $\ \ wa\text{-}initial = brw\text{-}initial \ \delta,$
 $\ \ wa\text{-}invar = brw\text{-}invar \ \delta$
 \rangle

lemma $brw\text{-}cond\text{-}abs: \ \Sigma \in brw\text{-}cond \ Qi \longleftrightarrow (brw\text{-}\alpha \ \Sigma) \in br'\text{-}cond \ Qi$
 $\langle \text{proof} \rangle$

lemma $brw\text{-}initial\text{-}abs: \ \Sigma \in brw\text{-}initial \ \delta \Longrightarrow brw\text{-}\alpha \ \Sigma \in br'\text{-}initial \ \delta$
 $\langle \text{proof} \rangle$

lemma $brw\text{-}invar\text{-}initial: \ brw\text{-}initial \ \delta \subseteq brw\text{-}invar\text{-}add \ \delta$
 $\langle \text{proof} \rangle$

lemma $brw\text{-}step\text{-}abs:$
 $\llbracket (\Sigma, \Sigma') \in brw\text{-}step \ \delta \rrbracket \Longrightarrow (brw\text{-}\alpha \ \Sigma, brw\text{-}\alpha \ \Sigma') \in br'\text{-}step \ \delta$
 $\langle \text{proof} \rangle$

lemma $brw\text{-}step\text{-}invar:$
assumes $FIN[simp]: \ \text{finite } \delta$
assumes $INV: \ \Sigma \in brw\text{-}invar\text{-}add \ \delta$ **and** $BR'INV: \ brw\text{-}\alpha \ \Sigma \in br'\text{-}invar \ \delta$
assumes $STEP: \ (\Sigma, \Sigma') \in brw\text{-}step \ \delta$
shows $\Sigma' \in brw\text{-}invar\text{-}add \ \delta$
 $\langle \text{proof} \rangle$

theorem $brw\text{-}pref\text{-}br': \ wa\text{-}precise\text{-}refine \ (brw\text{-}algo \ Qi \ \delta) \ (br'\text{-}algo \ Qi \ \delta) \ brw\text{-}\alpha$
 $\langle \text{proof} \rangle$

interpretation $brw\text{-}pref:$
 $wa\text{-}precise\text{-}refine \ brw\text{-}algo \ Qi \ \delta \quad br'\text{-}algo \ Qi \ \delta \quad brw\text{-}\alpha$

$\langle \text{proof} \rangle$

theorem *brw-while-algo*: $\text{finite } \delta \implies \text{while-algo } (\text{brw-algo } Qi \delta)$
 $\langle \text{proof} \rangle$

lemma *fst-brw- α* : $\text{fst } (\text{brw-}\alpha \ s) = \text{dom } (\text{fst } s)$
 $\langle \text{proof} \rangle$

theorem *brw-invar-final*:

$\forall sc. sc \in \text{wa-invar } (\text{brw-algo } Qi \delta) \wedge sc \notin \text{wa-cond } (\text{brw-algo } Qi \delta)$
 $\longrightarrow (Qi \cap \text{dom } (\text{fst } sc) = \{\}) = (Qi \cap \text{b-accessible } \delta = \{\})$
 $\wedge (\text{witness-prop } \delta (\text{fst } sc))$
 $\langle \text{proof} \rangle$

Implementing a Step *inductive-set* *brw-inner-step*

$:: ('Q, 'L) \text{ ta-rule} \Rightarrow (('Q, 'L) \text{ brw-state} \times ('Q, 'L) \text{ brw-state}) \text{ set}$

for r **where**

$\llbracket c = \text{the } (\text{rcm } r); \Sigma = (Q, W, \text{rcm}); \Sigma' = (Q', W', \text{rcm}')$
 $\text{if } c \leq 1 \wedge (\text{lhs } r) \notin \text{dom } Q \text{ then}$
 $\quad Q' = Q(\text{lhs } r \mapsto \text{construct-witness } Q \ r)$
 $\text{else } Q' = Q;$
 $\text{if } c \leq 1 \wedge (\text{lhs } r) \notin \text{dom } Q \text{ then}$
 $\quad W' = \text{insert } (\text{lhs } r) \ W$
 $\text{else } W' = W;$
 $\text{rcm}' = \text{rcm } (r \mapsto (c - (1 :: \text{nat})))$
 $\rrbracket \implies (\Sigma, \Sigma') \in \text{brw-inner-step } r$

definition *brw-inner-invar*

$:: ('Q, 'L) \text{ ta-rule set} \Rightarrow 'Q \Rightarrow ('Q, 'L) \text{ brw-state} \Rightarrow ('Q, 'L) \text{ ta-rule set}$
 $\Rightarrow ('Q, 'L) \text{ brw-state} \Rightarrow \text{bool}$

where

brw-inner-invar rules $q == \lambda(Q, W, \text{rcm}) \text{ it } (Q', W', \text{rcm}')$.
(*br'-inner-invar* rules q (*brw- α* (Q, W, rcm)) *it* (*brw- α* (Q', W', rcm')) \wedge
 $(Q' \upharpoonright \text{dom } Q = Q) \wedge$
(*let* $\text{dsqr} = \{ r \in \text{rules} - \text{it. the } (\text{rcm } r) \leq 1 \}$ *in*
 $(\forall q \ t. Q' \ q = \text{Some } t \longrightarrow (Q \ q = \text{Some } t$
 $\vee (Q \ q = \text{None} \wedge (\exists r \in \text{dsqr. } q = \text{lhs } r \wedge t = \text{construct-witness } Q \ r))$
 $)$
 $)))$

lemma *brw-inner-step-abs*:

$(\Sigma, \Sigma') \in \text{brw-inner-step } r \implies \text{br}'\text{-inner-step } r \ (\text{brw-}\alpha \ \Sigma) = \text{brw-}\alpha \ \Sigma'$
 $\langle \text{proof} \rangle$

lemma *brw-inner-invar-imp-final*:

$\llbracket q \in W; \text{brw-inner-invar } \{r \in \delta. q \in \text{set } (\text{rhs } r)\} \ q \ (Q, W - \{q\}, \text{rcm}) \ \{\} \ \Sigma' \rrbracket$
 $\implies ((Q, W, \text{rcm}), \Sigma') \in \text{brw-step } \delta$

<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*: $brw\text{-inner-invar } \{r \in \delta. q \in set (rhsq r)\} q (Q, W - \{q\}, rcm) it \Sigma h$

assumes *STEP*: $(\Sigma h, \Sigma') \in brw\text{-inner-step } r$

shows $brw\text{-inner-invar } \{r \in \delta. q \in set (rhsq r)\} q (Q, W - \{q\}, rcm) (it - \{r\}) \Sigma'$

<proof>

lemma *brw-inner-invar-initial*:

$\llbracket q \in W \rrbracket \implies brw\text{-inner-invar } \{r \in \delta. q \in set (rhsq r)\} q (Q, W - \{q\}, rcm)$
 $\{r \in \delta. q \in set (rhsq r)\} (Q, W - \{q\}, rcm)$

<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*: $set\text{-iteratei } \alpha \text{ invar } iteratei$

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*:

$!! it r \Sigma. \llbracket r \in it; it \subseteq \{r \in \delta. q \in set (rhsq r)\}; cinvar it \Sigma \rrbracket$
 $\implies cinvar (it - \{r\}) (cstep r \Sigma)$

assumes *step-desc*:

$!! it r \Sigma. \llbracket r \in it; it \subseteq \{r \in \delta. q \in set (rhsq r)\}; cinvar it \Sigma \rrbracket$
 $\implies (\alpha s \Sigma, \alpha s (cstep r \Sigma)) \in brw\text{-inner-step } r$

assumes *it-set-desc*: $invar it\text{-set} \quad \alpha \text{ it-set} = \{r \in \delta. q \in set (rhsq r)\}$

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

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

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

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

<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* \times *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*:
 $\forall s. s \in \text{wa-invar } (\text{frp-algo } T1 \ T2) \wedge s \notin \text{wa-cond } (\text{frp-algo } T1 \ T2)$
 $\longrightarrow (\text{case } s \text{ of } (Q, W, \delta d) \Rightarrow$
 ($\mid \text{ta-initial} = \text{ta-initial } T1 \times \text{ta-initial } T2,$
 $\text{ta-rules} = \delta d$
 $\mid) = \text{ta-fwd-reduce } (\text{ta-prod } T1 \ T2))$
<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 efficient 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* \Rightarrow *hashcode*
where
hashcode-of-ta-rule (*q* \rightarrow *f* *qs*) = *hashcode* *q* + *hashcode* *f* + *hashcode* *qs*
definition [*simp*]: *hashcode* = *hashcode-of-ta-rule*

definition *def-hashmap-size::('a,'b) ta-rule itself \Rightarrow 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 \Rightarrow 2 * hashcode a | USW2 b \Rightarrow 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-iftl-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*
 $:: (('q, 'l) \text{ ta-rule} \Rightarrow 'i::\text{hashable}) \Rightarrow ('q, 'l) \text{ ta-rule } ls$
 $\Rightarrow ('i, ('q, 'l) \text{ ta-rule } ls) \text{ hm}$
where *build-rule-index* == *hll-idx.idx-build*

definition *build-rule-index-f* δ == *build-rule-index* ($\lambda r. \text{rhsl } r$) δ

definition *build-rule-index-s* δ == *build-rule-index* ($\lambda r. \text{lhs } r$) δ

definition *build-rule-index-sf* δ == *build-rule-index* ($\lambda r. (\text{lhs } r, \text{rhsl } r)$) 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 ($\lambda r. (\text{lhs } r, \text{rhsl } r)$) (*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
 $hta-idx-sf :: ('q \times 'l,('q,'l) ta-rule ls) hm option$

— Abstraction of a concrete tree automaton to an abstract one

definition $hta-\alpha$

where $hta-\alpha H = \langle ta-initial = hs-\alpha (hta-Qi H), ta-rules = ls-\alpha (hta-\delta H) \rangle$

— Builds the f-index if not present

definition $hta-ensure-idx-f H ==$

case $hta-idx-f H$ of

$None \Rightarrow H \langle hta-idx-f := Some (build-rule-index-f (hta-\delta H)) \rangle$ |

$Some - \Rightarrow H$

— Builds the s-index if not present

definition $hta-ensure-idx-s H ==$

case $hta-idx-s H$ of

$None \Rightarrow H \langle hta-idx-s := Some (build-rule-index-s (hta-\delta H)) \rangle$ |

$Some - \Rightarrow H$

— Builds the sf-index if not present

definition $hta-ensure-idx-sf H ==$

case $hta-idx-sf H$ of

$None \Rightarrow H \langle hta-idx-sf := Some (build-rule-index-sf (hta-\delta H)) \rangle$ |

$Some - \Rightarrow H$

lemma $hta-ensure-idx-f-correct-\alpha[simp]$:

$hta-\alpha (hta-ensure-idx-f H) = hta-\alpha H$

<proof>

lemma $hta-ensure-idx-s-correct-\alpha[simp]$:

$hta-\alpha (hta-ensure-idx-s H) = hta-\alpha H$

<proof>

lemma $hta-ensure-idx-sf-correct-\alpha[simp]$:

$hta-\alpha (hta-ensure-idx-sf H) = hta-\alpha H$

<proof>

lemma $hta-ensure-idx-other[simp]$:

$hta-Qi (hta-ensure-idx-f H) = hta-Qi H$

$hta-\delta (hta-ensure-idx-f H) = hta-\delta H$

$hta-Qi (hta-ensure-idx-s H) = hta-Qi H$

$hta-\delta (hta-ensure-idx-s H) = hta-\delta H$

$hta-Qi (hta-ensure-idx-sf H) = hta-Qi H$

$hta-\delta (hta-ensure-idx-sf H) = hta-\delta H$

<proof>

definition $hta\text{-has-idx-f } H == hta\text{-idx-f } H \neq \text{None}$

— Check whether the s-index is present

definition $hta\text{-has-idx-s } H == hta\text{-idx-s } H \neq \text{None}$

— Check whether the sf-index is present

definition $hta\text{-has-idx-sf } H == hta\text{-idx-sf } H \neq \text{None}$

lemma $hta\text{-idx-f-pres}$

$[simp, intro!]: hta\text{-has-idx-f } (hta\text{-ensure-idx-f } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-s } H \implies hta\text{-has-idx-s } (hta\text{-ensure-idx-f } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-sf } H \implies hta\text{-has-idx-sf } (hta\text{-ensure-idx-f } H)$

$\langle proof \rangle$

lemma $hta\text{-idx-s-pres}$

$[simp, intro!]: hta\text{-has-idx-s } (hta\text{-ensure-idx-s } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-f } H \implies hta\text{-has-idx-f } (hta\text{-ensure-idx-s } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-sf } H \implies hta\text{-has-idx-sf } (hta\text{-ensure-idx-s } H)$

$\langle proof \rangle$

lemma $hta\text{-idx-sf-pres}$

$[simp, intro!]: hta\text{-has-idx-sf } (hta\text{-ensure-idx-sf } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-f } H \implies hta\text{-has-idx-f } (hta\text{-ensure-idx-sf } H) \text{ and}$

$[simp, intro]: hta\text{-has-idx-s } H \implies hta\text{-has-idx-s } (hta\text{-ensure-idx-sf } H)$

$\langle proof \rangle$

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

definition $hta\text{-lookup-f } f H == hll\text{-idx.lookup } f (the (hta\text{-idx-f } H))$

— Lookup rules by lhs-state

definition $hta\text{-lookup-s } q H == hll\text{-idx.lookup } q (the (hta\text{-idx-s } H))$

— Lookup rules by function symbol and lhs-state

definition $hta\text{-lookup-sf } q f H == hll\text{-idx.lookup } (q,f) (the (hta\text{-idx-sf } H))$

— 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 $invar[simp, intro!]:$

$hs\text{-invar } (hta\text{-}Q_i H)$

$ls\text{-invar } (hta\text{-}\delta H)$

— The indices are correct, if present

assumes $index\text{-correct}:$

$hta\text{-idx-f } H = \text{Some } idx\text{-f}$

$\implies hll\text{-idx.is-index } rhsl (ls\text{-}\alpha (hta\text{-}\delta H)) \text{ } idx\text{-f}$

$hta\text{-idx-s } H = \text{Some } idx\text{-s}$

$\implies hll\text{-idx.is-index } lhs (ls\text{-}\alpha (hta\text{-}\delta H)) \text{ } idx\text{-s}$

$hta\text{-idx-sf } H = \text{Some } idx\text{-sf}$

$\implies \text{hll-idx.is-index } (\lambda r. (\text{lhs } r, \text{rhsl } r)) (\text{ls-}\alpha (\text{hta-}\delta H)) \text{ idx-sf}$

begin

— Inside this locale, some shorthand notations for the sets of rules and initial states are used

abbreviation $\delta == \text{hta-}\delta H$

abbreviation $Qi == \text{hta-}Qi H$

— The lookup-xxx operations are correct

lemma *hta-lookup-f-correct*:

$\text{hta-has-idx-f } H \implies \text{ls-}\alpha (\text{hta-lookup-f } f H) = \{r \in \text{ls-}\alpha \delta . \text{rhsl } r = f\}$

$\text{hta-has-idx-f } H \implies \text{ls-invar } (\text{hta-lookup-f } f H)$

<proof>

lemma *hta-lookup-s-correct*:

$\text{hta-has-idx-s } H \implies \text{ls-}\alpha (\text{hta-lookup-s } q H) = \{r \in \text{ls-}\alpha \delta . \text{lhs } r = q\}$

$\text{hta-has-idx-s } H \implies \text{ls-invar } (\text{hta-lookup-s } q H)$

<proof>

lemma *hta-lookup-sf-correct*:

$\text{hta-has-idx-sf } H$

$\implies \text{ls-}\alpha (\text{hta-lookup-sf } q f H) = \{r \in \text{ls-}\alpha \delta . \text{lhs } r = q \wedge \text{rhsl } r = f\}$

$\text{hta-has-idx-sf } H \implies \text{ls-invar } (\text{hta-lookup-sf } q f H)$

<proof>

lemma *hta-ensure-idx-f-correct*[*simp, intro!*]: $\text{hashedTa } (\text{hta-ensure-idx-f } H)$

<proof>

lemma *hta-ensure-idx-s-correct*[*simp, intro!*]: $\text{hashedTa } (\text{hta-ensure-idx-s } H)$

<proof>

lemma *hta-ensure-idx-sf-correct*[*simp, intro!*]: $\text{hashedTa } (\text{hta-ensure-idx-sf } H)$

<proof>

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

lemma *hta- α -is-ta*[*simp, intro!*]: $\text{tree-automaton } (\text{hta-}\alpha H)$

<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* $Qi \delta ==$

$(\mid \text{hta-}Qi = Qi,$

```

    hta- $\delta$  =  $\delta$ ,
    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 ($\lambda Q q. q \in Q$) 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 ( $\lambda r. \text{case } r \text{ of } (q \rightarrow f' qs) \Rightarrow$ 
      if list-all-zip ( $\lambda Q q. \text{ls-memb } q Q$ ) 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 == $\neg \text{lh-set-xx.g-disjoint (faccs' H t) (hta-Qi H)}$*

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- $\alpha$  (faccs' H t) = faccs (ls- $\alpha$  (hta- $\delta$  H)) t (is ?T1)

```

$List.map\ ls-\alpha\ (List.map\ (faccs'\ H)\ ts)$
 $=\ List.map\ (faccs\ (ls-\alpha\ (hta-\delta\ H)))\ ts\ (\mathbf{is}\ ?T2)$

$\langle proof \rangle$

lemma *hta-mem'-correct*:

$hta-has-idx-f\ H\ \Longrightarrow\ hta-mem'\ t\ H\ \longleftrightarrow\ t \in ta-lang\ (hta-\alpha\ H)$

$\langle proof \rangle$

theorem *hta-mem-correct*: $hta-mem\ t\ H\ \longleftrightarrow\ t \in ta-lang\ (hta-\alpha\ H)$

$\langle proof \rangle$

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*

$::\ ('q1::hashable, 'l::hashable)\ ta-rule\ ls$
 $\Rightarrow\ ('q2::hashable, 'l)\ ta-rule\ ls\ \Rightarrow\ ('q1 \times 'q2, 'l)\ ta-rule\ ls$

where *δ -prod-h $\delta 1\ \delta 2$* ==

$lll-ift-cp.inj-image-filter-cp\ (\lambda(r1,r2).\ r-prod\ r1\ r2)$

$(\lambda(r1,r2).\ rhsl\ r1 = rhsl\ r2$

$\wedge\ length\ (rhsq\ r1) = length\ (rhsq\ r2))$

$\delta 1\ \delta 2$

lemma *r-prod-inj*:

$\llbracket\ rhsl\ r1 = rhsl\ r2;\ length\ (rhsq\ r1) = length\ (rhsq\ r2);$
 $rhsl\ r1' = rhsl\ r2'; length\ (rhsq\ r1') = length\ (rhsq\ r2');$
 $r-prod\ r1\ r2 = r-prod\ r1'\ r2'\ \rrbracket\ \Longrightarrow\ r1=r1' \wedge r2=r2'$

$\langle proof \rangle$

lemma *δ -prod-h-correct*:

assumes *INV[simp]*: $ls-invar\ \delta 1\ \quad ls-invar\ \delta 2$

shows

$ls-\alpha\ (\delta-prod-h\ \delta 1\ \delta 2) = \delta-prod\ (ls-\alpha\ \delta 1)\ (ls-\alpha\ \delta 2)$

$ls-invar\ (\delta-prod-h\ \delta 1\ \delta 2)$

$\langle proof \rangle$

definition *hta-prodWR $H1\ H2$* ==

$init-hta\ (hhh-cart.cart\ (hta-Qi\ H1)\ (hta-Qi\ H2))\ (\delta-prod-h\ (hta-\delta\ H1)\ (hta-\delta\ H2))$

lemma *hta-prodWR-correct-aux*:

assumes *A*: $hashedTa\ H1\ \quad hashedTa\ H2$

shows

$hta-\alpha (hta-prodWR H1 H2) = ta-prod (hta-\alpha H1) (hta-\alpha H2)$ (is ?T1)
 $hashedTa (hta-prodWR H1 H2)$ (is ?T2)
 <proof>

lemma *hta-prodWR-correct*:
assumes *TA*: $hashedTa H1$ $hashedTa H2$
shows
 $ta-lang (hta-\alpha (hta-prodWR H1 H2))$
 $= ta-lang (hta-\alpha H1) \cap ta-lang (hta-\alpha H2)$
 $hashedTa (hta-prodWR H1 H2)$
 <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*
 $= (q1 \times q2) hs \times (q1 \times q2) list \times (q1 \times q2, l) ta-rule ls$

— Abstraction mapping to algorithm specified in Section 4.

definition *pa- α*
 $:: (q1::hashable, q2::hashable, l::hashable) pa-state$
 $\Rightarrow (q1, q2, l) frp-state$
where *pa- α* $S == let (Q, W, \delta d) = S in (hs-\alpha Q, W, ls-\alpha \delta d)$

definition *pa-cond*
 $:: (q1::hashable, q2::hashable, l::hashable) pa-state \Rightarrow bool$
where *pa-cond* $S == let (Q, W, \delta d) = S in W \neq []$

— Adds all successor states to the set of discovered states and to the worklist

fun *pa-upd-rule*
 $:: (q1 \times q2) hs \Rightarrow (q1 \times q2) list$
 $\Rightarrow ((q1::hashable) \times (q2::hashable)) list$
 $\Rightarrow ((q1 \times q2) hs \times (q1 \times q2) list)$
where
pa-upd-rule $Q W [] = (Q, W) |$
pa-upd-rule $Q W (qp \# qs) = ($
 $if \neg hs-memb qp Q then$
 $pa-upd-rule (hs-ins qp Q) (qp \# W) qs$
 $else pa-upd-rule Q W qs$
 $)$

definition *pa-step*
 $:: (q1::hashable, l::hashable) hashedTa$
 $\Rightarrow (q2::hashable, l) hashedTa$
 $\Rightarrow (q1, q2, l) pa-state \Rightarrow (q1, q2, l) pa-state$
where *pa-step* $H1 H2 S == let$
 $(Q, W, \delta d) = S;$


```

(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)

$\exists Wn. \text{distinct } Wn \wedge \text{set } Wn = \text{set } qs - \text{hs-}\alpha Q \wedge W' = Wn @ W \text{ (is ?T3)}$

<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*: $(Q, W, \delta d) \in \text{pa-invar } H1 \ H2$
assumes *COND*: *pa-cond (Q, W, δd)*
shows
 $(\text{pa-step } H1 \ H2 \ (Q, W, \delta d)) \in \text{pa-invar-add } (\text{is } ?T1)$
 $(\text{pa-}\alpha \ (Q, W, \delta d), \text{pa-}\alpha \ (\text{pa-step } H1 \ H2 \ (Q, W, \delta d)))$
 $\in \text{frp-step } (\text{ls-}\alpha \ (\text{hta-}\delta \ H1)) \ (\text{ls-}\alpha \ (\text{hta-}\delta \ H2)) \ (\text{is } ?T2)$
 $\langle \text{proof} \rangle$

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* $(\text{det-wa-wa } (\text{pa-det-algo } H1 \ H2))$
 $(\text{frp-algo } (\text{hta-}\alpha \ H1) \ (\text{hta-}\alpha \ H2))$
 $\text{pa-}\alpha$
 $\langle \text{proof} \rangle$

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* $(\text{det-wa-wa } (\text{pa-det-algo } H1 \ H2))$
 $\langle \text{proof} \rangle$

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 $(Q, W, \delta d) = \text{while pa-cond } (\text{pa-step } H1 \ H2) \ (\text{pa-initial } H1 \ H2)$ *in*
 $\text{init-hta } (\text{hhh-cart.cart } (\text{hta-}Qi \ H1) \ (\text{hta-}Qi \ H2)) \ \delta d$

definition *hta-prod* *H1 H2* ==

$\text{hta-prod}' \ (\text{hta-ensure-idx-s } H1) \ (\text{hta-ensure-idx-sf } H2)$

lemma *hta-prod'-correct-aux*:

assumes *TA*: *hashedTa H1 hashedTa H2*
assumes *idx*: *hta-has-idx-s H1 hta-has-idx-sf H2*

shows $hta-\alpha$ ($hta-prod'$ $H1$ $H2$)
 $= ta-fwd-reduce$ ($ta-prod$ ($hta-\alpha$ $H1$) ($hta-\alpha$ $H2$)) (**is** $?T1$)
 $hashedTa$ ($hta-prod'$ $H1$ $H2$) (**is** $?T2$)
 $\langle proof \rangle$

theorem $hta-prod'$ -correct:
assumes TA : $hashedTa$ $H1$ $hashedTa$ $H2$
assumes HI : $hta-has-idx-s$ $H1$ $hta-has-idx-sf$ $H2$
shows
 $ta-lang$ ($hta-\alpha$ ($hta-prod'$ $H1$ $H2$))
 $= ta-lang$ ($hta-\alpha$ $H1$) $\cap ta-lang$ ($hta-\alpha$ $H2$)

 $hashedTa$ ($hta-prod'$ $H1$ $H2$)
 $\langle proof \rangle$

lemma $hta-prod$ -correct-aux:
assumes $TA[simp]$: $hashedTa$ $H1$ $hashedTa$ $H2$
shows
 $hta-\alpha$ ($hta-prod$ $H1$ $H2$) $= ta-fwd-reduce$ ($ta-prod$ ($hta-\alpha$ $H1$) ($hta-\alpha$ $H2$))
 $hashedTa$ ($hta-prod$ $H1$ $H2$)
 $\langle proof \rangle$

theorem $hta-prod$ -correct:
assumes TA : $hashedTa$ $H1$ $hashedTa$ $H2$
shows
 $ta-lang$ ($hta-\alpha$ ($hta-prod$ $H1$ $H2$))
 $= ta-lang$ ($hta-\alpha$ $H1$) $\cap ta-lang$ ($hta-\alpha$ $H2$)
 $hashedTa$ ($hta-prod$ $H1$ $H2$)
 $\langle proof \rangle$

5.6 Remap States

definition $hta-remap$
 $:: ('q::hashable \Rightarrow 'qn::hashable) \Rightarrow ('q,'l::hashable) hashedTa$
 $\Rightarrow ('qn,'l) hashedTa$
where $hta-remap$ f $H ==$
 $init-hta$ ($hh-set-xy.g-image$ f ($hta-Qi$ H))
 $(ll-set-xy.g-image$ ($remap-rule$ f) ($hta-\delta$ H))

lemma (**in** $hashedTa$) $hta-remap$ -correct:
shows $hta-\alpha$ ($hta-remap$ f H) $= ta-remap$ f ($hta-\alpha$ H)
 $hashedTa$ ($hta-remap$ f H)
 $\langle proof \rangle$

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 → 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
    (λr. Some (rule-states-l r)) (hta-δ H)))

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

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

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

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

declare hta-reindex-def [code del]

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

  ⟨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*

$:: ('q1::hashable, 'l::hashable) hashedTa$
 $\Rightarrow ('q2::hashable, 'l) hashedTa$
 $\Rightarrow (('q1, 'q2) ustate-wrapper, 'l) hashedTa$
where *hta-union* $H1 H2 ==$
 $init-hta (hs-union (hh-set-xy.g-image USW1 (hta-Qi H1))$
 $(hh-set-xy.g-image USW2 (hta-Qi H2)))$
 $(ls-union-dj (ll-set-xy.g-image (remap-rule USW1) (hta-\delta H1))$
 $(ll-set-xy.g-image (remap-rule USW2) (hta-\delta H2)))$

lemma *hta-union-correct'*:

assumes *TA*: $hashedTa H1 \quad hashedTa H2$
shows $hta-\alpha (hta-union H1 H2)$
 $= ta-union-wrap (hta-\alpha H1) (hta-\alpha H2) \text{ (is ?T1)}$
 $hashedTa (hta-union H1 H2) \text{ (is ?T2)}$

<proof>

theorem *hta-union-correct*:

assumes *TA*: $hashedTa H1 \quad hashedTa H2$
shows
 $ta-lang (hta-\alpha (hta-union H1 H2))$
 $= ta-lang (hta-\alpha H1) \cup ta-lang (hta-\alpha H2) \text{ (is ?T1)}$
 $hashedTa (hta-union H1 H2) \text{ (is ?T2)}$

<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* $:: unit \Rightarrow ('q::hashable, 'l::hashable) hashedTa$

where *hta-empty* $u == init-hta (hs-empty ()) (ls-empty ())$

lemma *hta-empty-correct* [*simp*, *intro!*]:

shows $(hta-\alpha (hta-empty ())) = ta-empty$
 $hashedTa (hta-empty ())$

<proof>

definition *hta-add-qi*

$:: 'q \Rightarrow ('q::hashable, 'l::hashable) hashedTa \Rightarrow ('q, 'l) hashedTa$
where *hta-add-qi* $qi H == init-hta (hs-ins qi (hta-Qi H)) (hta-\delta H)$

lemma (**in** *hashedTa*) *hta-add-qi-correct* [*simp*, *intro!*]:

shows $hta-\alpha (hta-add-qi qi H)$
 $= (| ta-initial = insert qi (ta-initial (hta-\alpha H)),$

$$\begin{array}{l}
ta\text{-rules} = ta\text{-rules } (hta\text{-}\alpha H) \\
\Downarrow \\
hashedTa (hta\text{-add-qi } qi H) \\
\langle proof \rangle
\end{array}$$

lemmas $[simp, intro] = hashedTa.hta\text{-add-qi-correct}$

— Add a rule to the automaton

definition *hta-add-rule*

$$\begin{array}{l}
:: ('q, 'l) ta\text{-rule} \Rightarrow ('q::hashable, 'l::hashable) hashedTa \\
\Rightarrow ('q, 'l) hashedTa \\
\text{where } hta\text{-add-rule } r H == init\text{-hta } (hta\text{-Qi } H) (ls\text{-ins } r (hta\text{-}\delta H))
\end{array}$$

lemma (in *hashedTa*) *hta-add-rule-correct* $[simp, intro!]$:

$$\begin{array}{l}
\text{shows } hta\text{-}\alpha (hta\text{-add-rule } r H) \\
= (\Downarrow ta\text{-initial} = ta\text{-initial } (hta\text{-}\alpha H), \\
ta\text{-rules} = insert\ r (ta\text{-rules } (hta\text{-}\alpha H)) \\
\Downarrow \\
hashedTa (hta\text{-add-rule } r H) \\
\langle proof \rangle
\end{array}$$

lemmas $[simp, intro] = hashedTa.hta\text{-add-rule-correct}$

— Reduces an automaton to the given set of states

definition *hta-reduce* $H Q ==$

$$\begin{array}{l}
init\text{-hta } (hs\text{-inter } Q (hta\text{-Qi } H)) \\
(ll\text{-set-xy.g-image-filter} \\
(\lambda r. \text{if } hs\text{-memb } (lhs\ r) Q \wedge list\text{-all } (\lambda q. hs\text{-memb } q Q) (rhsq\ r) \text{ then} \\
Some\ r \text{ else None}) \\
(hta\text{-}\delta H))
\end{array}$$

theorem (in *hashedTa*) *hta-reduce-correct*:

assumes $INV[simp]: hs\text{-invar } Q$

shows

$$hta\text{-}\alpha (hta\text{-reduce } H Q) = ta\text{-reduce } (hta\text{-}\alpha H) (hs\text{-}\alpha Q) \text{ (is ?T1)}$$

$$hashedTa (hta\text{-reduce } H Q) \text{ (is ?T2)}$$

$\langle proof \rangle$

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* $q r res ==$

$$\begin{array}{l}
\text{case } hm\text{-lookup } q \text{ res of} \\
None \Rightarrow hm\text{-update } q (ls\text{-ins } r (ls\text{-empty } ())) \text{ res} \mid \\
Some\ s \Rightarrow hm\text{-update } q (ls\text{-ins } r s) \text{ res}
\end{array}$$

— Lookup the set of rules with given state on rhs

definition *rqrm-lookup* $rqrm\ q == \text{case } hm\text{-lookup } q\ rqrm \text{ of}$
 $None \Rightarrow ls\text{-empty } () \mid$
 $Some\ s \Rightarrow s$

— Build the index from a set of rules

definition *build-rqrm*
 $:: ('q::hashable, 'l::hashable)\ ta\text{-rule } ls$
 $\Rightarrow ('q, ('q, 'l)\ ta\text{-rule } ls)\ hm$

where

build-rqrm $\delta ==$
 $ls\text{-iteratei } \delta\ (\lambda\text{-}. True)$
 $(\lambda r\ res.$
 $\quad foldl\ (\lambda res\ q.\ rqrm\text{-add } q\ r\ res)\ res\ (rhsq\ r)$
 $)$
 $(hm\text{-empty } ())$

— Whether the index satisfies the map and set invariants

definition *rqrm-invar* $rqrm ==$
 $hm\text{-invar } rqrm \wedge (\forall q.\ ls\text{-invar } (rqrm\text{-lookup } rqrm\ q))$

— Whether the index really maps a state to the set of rules with this state on their rhs

definition *rqrm-prop* $\delta\ rqrm ==$
 $\forall q.\ ls\text{-}\alpha\ (rqrm\text{-lookup } rqrm\ q) = \{r \in \delta.\ q \in \text{set } (rhsq\ r)\}$

lemma *rqrm- α -lookup-update[simp]*:

rqrm-invar $rqrm \implies$
 $ls\text{-}\alpha\ (rqrm\text{-lookup } (rqrm\text{-add } q\ r\ rqrm)\ q')$
 $= (\text{if } q=q' \text{ then}$
 $\quad insert\ r\ (ls\text{-}\alpha\ (rqrm\text{-lookup } rqrm\ q'))$
 else
 $\quad ls\text{-}\alpha\ (rqrm\text{-lookup } rqrm\ q')$
 $)$
 $\langle proof \rangle$

lemma *rqrm-propD*:

rqrm-prop $\delta\ rqrm \implies ls\text{-}\alpha\ (rqrm\text{-lookup } rqrm\ q) = \{r \in \delta.\ q \in \text{set } (rhsq\ r)\}$
 $\langle proof \rangle$

lemma *build-rqrm-correct*:

fixes δ

assumes [simp]: $ls\text{-invar } \delta$

shows *rqrm-invar* (*build-rqrm* δ) (**is** ?T1) **and**

rqrm-prop ($ls\text{-}\alpha\ \delta$) (*build-rqrm* δ) (**is** ?T2)

$\langle proof \rangle$

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 \Rightarrow ('Q,'L) br'-state
where *brc- α* == $\lambda(Q,W,rcm). (hs-\alpha\ Q, set\ W, hm-\alpha\ rcm)$

definition *brc-invar-add* :: ('Q::hashable,'L::hashable) brc-state set

where

brc-invar-add == $\{(Q,W,rcm).\}$

hs-invar Q \wedge

distinct W \wedge

hm-invar rcm

~~*hs-invar* Q~~

definition *brc-invar* δ == *brc-invar-add* \cap {s. *brc- α* s \in *br'-invar* δ }

definition *brc-cond* :: ('q::hashable,'l::hashable) brc-state \Rightarrow bool

where *brc-cond* == $\lambda(Q,W,rcm). W \neq []$

definition *brc-inner-step*

:: ('q,'l) ta-rule \Rightarrow ('q::hashable,'l::hashable) brc-state

\Rightarrow ('q,'l) brc-state

where

brc-inner-step r == $\lambda(Q,W,rcm).$

let c=the (hm-lookup r rcm);

rcm' = hm-update r (c-(1::nat)) rcm;

Q' = (if c \leq 1 then hs-ins (lhs r) Q else Q);

W' = (if c \leq 1 \wedge \neg 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

\Rightarrow ('q::hashable,'l::hashable) brc-state

\Rightarrow ('q,'l) brc-state

where

brc-step rqr == $\lambda(Q,W,rcm).$

ls-iteratei (rqr-lookup rqr (hd W)) ($\lambda-. True$) *brc-inner-step*
(Q,tl W, rcm)

— Initial concrete state

definition *brc-ig* :: ('q,'l) ta-rule ls \Rightarrow 'q::hashable hs

where *brc-ig* δ == lh-set-xy.g-image-filter ($\lambda r.$

if rhsq r = [] then Some (lhs r) else None) δ

definition *brc-rcm-init*

:: ('q::hashable,'l::hashable) ta-rule ls

\Rightarrow (('q,'l) ta-rule,nat) hm

where *brc-rcm-init* δ ==
ls-iteratei δ (λ -. *True*)
 $(\lambda r$ *res*. *hm-update* r $((\text{length } (\text{remdups } (\text{rhsq } r))))$) *res*)
(hm-empty $()$)

definition *brc-initial*
 $:: ('q::\text{hashable}, 'l::\text{hashable})$ *ta-rule* $ls \Rightarrow ('q, 'l)$ *brc-state*
where *brc-initial* δ ==
 $\text{let } iq = \text{brc-iq } \delta$ *in*
 $(iq, \text{hs-to-list } (iq), \text{brc-rcm-init } \delta)$

definition *brc-det-algo* $rqrm$ δ == (
dwa-cond = *brc-cond*,
dwa-step = *brc-step* $rqrm$,
dwa-initial = *brc-initial* δ ,
dwa-invar = *brc-invar* $(ls-\alpha \delta)$
 $)$

— Additional facts needed from the abstract level

lemma *brc-inv-imp-WssQ*: $\text{brc-}\alpha (Q, W, \text{rcm}) \in \text{br}'\text{-invar } \delta \implies \text{set } W \subseteq \text{hs-}\alpha Q$
 $\langle \text{proof} \rangle$

lemma *brc-iq-correct*:
assumes [*simp*]: *ls-invar* δ
shows *hs-invar* $(\text{brc-iq } \delta)$
 $\text{hs-}\alpha (\text{brc-iq } \delta) = \text{br-iq } (ls-\alpha \delta)$
 $\langle \text{proof} \rangle$

lemma *brc-rcm-init-correct*:
assumes *INV*[*simp*]: *ls-invar* δ
shows $r \in ls-\alpha \delta$
 $\implies \text{hm-}\alpha (\text{brc-rcm-init } \delta) r = \text{Some } ((\text{card } (\text{set } (\text{rhsq } r))))$
(is - \implies ?T1 r) and
 $\text{hm-invar } (\text{brc-rcm-init } \delta)$ **(is ?T2)**
 $\langle \text{proof} \rangle$

lemma *brc-inner-step-br'-desc*:
 $\llbracket (Q, W, \text{rcm}) \in \text{brc-invar } \delta \rrbracket \implies \text{brc-}\alpha (\text{brc-inner-step } r (Q, W, \text{rcm})) = (
\text{if the } (\text{hm-}\alpha \text{rcm } r) \leq 1 \text{ then}$
 $\text{insert } (\text{lhs } r) (\text{hs-}\alpha Q)$
 $\text{else } \text{hs-}\alpha Q,$
 $\text{if the } (\text{hm-}\alpha \text{rcm } r) \leq 1 \wedge (\text{lhs } r) \notin \text{hs-}\alpha Q \text{ then}$
 $\text{insert } (\text{lhs } r) (\text{set } W)$
 $\text{else } (\text{set } W),$
 $((\text{hm-}\alpha \text{rcm})(r \mapsto \text{the } (\text{hm-}\alpha \text{rcm } r) - 1))$
 $)$
 $\langle \text{proof} \rangle$

lemma *brc-step-invar*:

assumes $RQRM$: $rqrm\text{-invar } rqrm$
shows $\llbracket \Sigma \in brc\text{-invar-add}; brc\text{-}\alpha \Sigma \in br'\text{-invar } \delta; brc\text{-cond } \Sigma \rrbracket$
 $\implies (brc\text{-step } rqrm \Sigma) \in brc\text{-invar-add}$
 $\langle proof \rangle$

lemma $brc\text{-step-abs}$:
assumes $RQRM$: $rqrm\text{-invar } rqrm \quad rqrm\text{-prop } \delta \quad rqrm$
assumes A : $\Sigma \in brc\text{-invar } \delta \quad brc\text{-cond } \Sigma$
shows $(brc\text{-}\alpha \Sigma, brc\text{-}\alpha (brc\text{-step } rqrm \Sigma)) \in br'\text{-step } \delta$
 $\langle proof \rangle$

lemma $brc\text{-initial-invar}$: $ls\text{-invar } \delta \implies (brc\text{-initial } \delta) \in brc\text{-invar-add}$
 $\langle proof \rangle$

lemma $brc\text{-cond-abs}$: $brc\text{-cond } \Sigma \longleftrightarrow (brc\text{-}\alpha \Sigma) \in br'\text{-cond}$
 $\langle proof \rangle$

lemma $brc\text{-initial-abs}$:
 $ls\text{-invar } \delta \implies brc\text{-}\alpha (brc\text{-initial } \delta) \in br'\text{-initial } (ls\text{-}\alpha \delta)$
 $\langle proof \rangle$

lemma $brc\text{-pref-br}'$:
assumes $RQRM[simp]$: $rqrm\text{-invar } rqrm \quad rqrm\text{-prop } (ls\text{-}\alpha \delta) \quad rqrm$
assumes $INV[simp]$: $ls\text{-invar } \delta$
shows $wa\text{-precise-refine } (det\text{-wa-wa } (brc\text{-det-algo } rqrm \delta))$
 $(br'\text{-algo } (ls\text{-}\alpha \delta))$
 $brc\text{-}\alpha$
 $\langle proof \rangle$

lemma $brc\text{-while-algo}$:
assumes $RQRM[simp]$: $rqrm\text{-invar } rqrm \quad rqrm\text{-prop } (ls\text{-}\alpha \delta) \quad rqrm$
assumes $INV[simp]$: $ls\text{-invar } \delta$
shows $while\text{-algo } (det\text{-wa-wa } (brc\text{-det-algo } rqrm \delta))$
 $\langle proof \rangle$

lemmas $brc\text{-det-while-algo} =$
 $det\text{-while-algo-intro}[OF \ brc\text{-while-algo}]$

lemma $fst\text{-brc-}\alpha$: $fst (brc\text{-}\alpha s) = hs\text{-}\alpha (fst s)$
 $\langle proof \rangle$

lemmas $brc\text{-invar-final} =$
 $wa\text{-precise-refine.transfer-correctness}[OF$
 $brc\text{-pref-br}' \ br'\text{-invar-final, unfolded } fst\text{-brc-}\alpha]$

definition $hta\text{-bwd-reduce } H ==$
 $let \ rqrm = build\text{-rqrm } (hta\text{-}\delta \ H) \ in$

$hta\text{-}reduce$
 H
 $(fst (while\ brc\text{-}cond (brc\text{-}step\ rqrm) (brc\text{-}initial (hta\text{-}\delta\ H))))$

theorem (in *hashedTa*) *hta-bwd-reduce-correct*:
shows $hta\text{-}\alpha (hta\text{-}bwd\text{-}reduce\ H)$
 $= ta\text{-}reduce (hta\text{-}\alpha\ H) (b\text{-}accessible (ls\text{-}\alpha (hta\text{-}\delta\ H)))$ (is ?T1)
 $hashedTa (hta\text{-}bwd\text{-}reduce\ H)$ (is ?T2)
 <proof>

5.9.1 Emptiness Check with Witness Computation

definition *brec-construct-witness*
 $:: ('q::hashable, 'l::hashable\ tree)\ hm \Rightarrow ('q, 'l)\ ta\text{-}rule \Rightarrow 'l\ tree$
where *brec-construct-witness* $Qm\ r ==$
 $NODE (rhsl\ r) (List.map (\lambda q. the (hm\text{-}lookup\ q\ Qm)) (rhsq\ r))$

lemma *brec-construct-witness-correct*:
 $\llbracket hm\text{-}invar\ Qm \rrbracket \Longrightarrow$
 $brec\text{-}construct\text{-}witness\ Qm\ r = construct\text{-}witness (hm\text{-}\alpha\ Qm)\ r$
 <proof>

type-synonym $('Q, 'L)\ brec\text{-}state$
 $= (('Q, 'L)\ tree)\ hm$
 $\times 'Q\ fifo$
 $\times (('Q, 'L)\ ta\text{-}rule, nat)\ hm$
 $\times 'Q\ option$

— Abstractions

definition *brec- α*
 $:: ('Q::hashable, 'L::hashable)\ brec\text{-}state \Rightarrow ('Q, 'L)\ brw\text{-}state$
where *brec- α* $== \lambda(Q, W, rcm, f). (hm\text{-}\alpha\ Q, set (fifo\text{-}\alpha\ W), (hm\text{-}\alpha\ rcm))$

definition *brec-inner-step*
 $:: 'q\ hs \Rightarrow ('q, 'l)\ ta\text{-}rule$
 $\Rightarrow ('q::hashable, 'l::hashable)\ brec\text{-}state$
 $\Rightarrow ('q, 'l)\ brec\text{-}state$
where *brec-inner-step* $Qi\ r == \lambda(Q, W, rcm, quit).$
 $let\ c = the (hm\text{-}lookup\ r\ rcm);$
 $cond = c \leq 1 \wedge hm\text{-}lookup (lhs\ r)\ Q = None;$
 $rcm' = hm\text{-}update\ r (c - (1::nat))\ rcm;$
 $Q' = (if\ cond\ then$
 $\quad hm\text{-}update (lhs\ r) (brec\text{-}construct\text{-}witness\ Q\ r)\ Q$
 $\quad else\ Q);$
 $W' = (if\ cond\ then\ fifo\text{-}enqueue (lhs\ r)\ W\ else\ W);$
 $quit' = (if\ c \leq 1 \wedge hs\text{-}memb (lhs\ r)\ Qi\ then\ Some (lhs\ r)\ else\ quit)$
 in

$(Q', W', rcm', qwit')$

definition *brec-step*

$:: ('q, ('q, 'l) \text{ ta-rule } ls) \text{ hm} \Rightarrow 'q \text{ hs}$
 $\Rightarrow ('q::\text{hashable}, 'l::\text{hashable}) \text{ brec-state}$
 $\Rightarrow ('q, 'l) \text{ brec-state}$
where *brec-step* *rgrm* *Qi* == $\lambda(Q, W, rcm, qwit)$.
 $\text{let } (q, W') = \text{fifo-dequeue } W \text{ in}$
 $ls\text{-iteratei } (rgrm\text{-lookup } rgrm \ q) \ (\lambda\text{-. } True)$
 $(\text{brec-inner-step } Qi) \ (Q, W', rcm, qwit)$

definition *brec-igq*

$:: ('q::\text{hashable}, 'l::\text{hashable}) \text{ ta-rule } ls \Rightarrow ('q, 'l \text{ tree}) \text{ hm}$
where *brec-igq* δ ==
 $ls\text{-iteratei } \delta \ (\lambda\text{-. } True) \ (\lambda r \ m. \text{ if } rhsq \ r = [] \text{ then}$
 $\quad \text{hm-update } (lhs \ r) \ (NODE \ (rhsl \ r) \ [])) \ m$
 $\text{else } m)$
 $(\text{hm-empty } ())$

definition *brec-initial*

$:: 'q \text{ hs} \Rightarrow ('q::\text{hashable}, 'l::\text{hashable}) \text{ ta-rule } ls$
 $\Rightarrow ('q, 'l) \text{ brec-state}$
where *brec-initial* *Qi* δ ==
 $\text{let } iq = \text{brc-ig} \ \delta \ \text{in}$
 $(\text{brec-igq } \delta,$
 $\text{hs-to-fifo.g-set-to-list } iq,$
 $\text{brc-rcm-init } \delta,$
 $\text{hh-set-xx.g-disjoint-witness } iq \ Qi)$

definition *brec-cond*

$:: ('q, 'l) \text{ brec-state} \Rightarrow \text{bool}$
where *brec-cond* == $\lambda(Q, W, rcm, qwit). \neg \text{fifo-isEmpty } W \wedge \text{qwit} = \text{None}$

definition *brec-invar-add*

$:: 'Q \text{ set} \Rightarrow ('Q::\text{hashable}, 'L::\text{hashable}) \text{ brec-state set}$
where
brec-invar-add *Qi* == $\{(Q, W, rcm, qwit).$
 $\text{hm-invar } Q \wedge$
 $\text{distinct } (\text{fifo-}\alpha \ W) \wedge$
 $\text{hm-invar } rcm \wedge$
 $(\text{case } qwit \text{ of}$
 $\quad \text{None} \Rightarrow Qi \cap \text{dom } (\text{hm-}\alpha \ Q) = \{\} \mid$
 $\quad \text{Some } q \Rightarrow q \in Qi \cap \text{dom } (\text{hm-}\alpha \ Q))\}$

definition *brec-invar* *Qi* δ == *brec-invar-add* *Qi* $\cap \{s. \text{brec-}\alpha \ s \in \text{brw-invar } \delta\}$

definition *brec-invar-inner* *Qi* ==

$brec\text{-invar-add } Qi \cap \{(Q, W, -, -). \text{ set } (fif\alpha W) \subseteq \text{ dom } (hm\text{-}\alpha Q)\}$

lemma *brec-invar-cons*:

$\Sigma \in brec\text{-invar } Qi \ \delta \implies \Sigma \in brec\text{-invar-inner } Qi$
 $\langle \text{proof} \rangle$

lemma *brec-brw-invar-cons*:

$brec\text{-}\alpha \ \Sigma \in brw\text{-invar } Qi \implies \text{ set } (fif\alpha (fst (snd \Sigma))) \subseteq \text{ dom } (hm\text{-}\alpha (fst \Sigma))$
 $\langle \text{proof} \rangle$

definition *brec-det-algo rqrM Qi δ ==* (

$dwa\text{-cond} = brec\text{-cond},$
 $dwa\text{-step} = brec\text{-step } rqrM \ Qi,$
 $dwa\text{-initial} = brec\text{-initial } Qi \ \delta,$
 $dwa\text{-invar} = brec\text{-invar } (hs\text{-}\alpha \ Qi) \ (ls\text{-}\alpha \ \delta)$

)

lemma *brec-igM-correct'*:

assumes $INV[simp]: ls\text{-invar } \delta$

shows

$\text{ dom } (hm\text{-}\alpha (brec\text{-igM } \delta)) = \{lhs \ r \mid r. r \in ls\text{-}\alpha \ \delta \wedge rhsq \ r = []\}$ (**is** ?T1)

$witness\text{-prop } (ls\text{-}\alpha \ \delta) \ (hm\text{-}\alpha (brec\text{-igM } \delta))$ (**is** ?T2)

$hm\text{-invar } (brec\text{-igM } \delta)$ (**is** ?T3)

$\langle \text{proof} \rangle$

lemma *brec-igM-correct*:

assumes $INV[simp]: ls\text{-invar } \delta$

shows $hm\text{-}\alpha (brec\text{-igM } \delta) \in brw\text{-iq } (ls\text{-}\alpha \ \delta)$

$\langle \text{proof} \rangle$

lemma *brec-inner-step-brw-desc*:

$\llbracket \Sigma \in brec\text{-invar-inner } (hs\text{-}\alpha \ Qi) \rrbracket$

$\implies (brec\text{-}\alpha \ \Sigma, brec\text{-}\alpha (brec\text{-inner-step } Qi \ r \ \Sigma)) \in brw\text{-inner-step } r$

$\langle \text{proof} \rangle$

lemma *brec-step-invar*:

assumes $RQRM: rqrM\text{-invar } rqrM \quad rqrM\text{-prop } \delta \ rqrM$

assumes $[simp]: hs\text{-invar } Qi$

shows $\llbracket \Sigma \in brec\text{-invar-add } (hs\text{-}\alpha \ Qi); brec\text{-}\alpha \ \Sigma \in brw\text{-invar } \delta; brec\text{-cond } \Sigma \rrbracket$

$\implies (brec\text{-step } rqrM \ Qi \ \Sigma) \in brec\text{-invar-add } (hs\text{-}\alpha \ Qi)$

$\langle \text{proof} \rangle$

lemma *brec-step-abs*:

assumes $RQRM: rqrM\text{-invar } rqrM \quad rqrM\text{-prop } \delta \ rqrM$

assumes $INV[simp]: hs\text{-invar } Qi$

assumes $A': \Sigma \in brec\text{-invar } (hs\text{-}\alpha \ Qi) \ \delta$

assumes $COND: brec\text{-cond } \Sigma$

shows $(brec\text{-}\alpha \ \Sigma, brec\text{-}\alpha (brec\text{-step } rqrM \ Qi \ \Sigma)) \in brw\text{-step } \delta$

$\langle proof \rangle$

lemma *brec-invar-initial*:

$\llbracket ls\text{-invar } \delta; hs\text{-invar } Qi \rrbracket \implies (brec\text{-initial } Qi \ \delta) \in brec\text{-invar-add } (hs\text{-}\alpha \ Qi)$
 $\langle proof \rangle$

lemma *brec-cond-abs*:

$\llbracket \Sigma \in brec\text{-invar } Qi \ \delta \rrbracket \implies brec\text{-cond } \Sigma \longleftrightarrow (brec\text{-}\alpha \ \Sigma) \in brw\text{-cond } Qi$
 $\langle proof \rangle$

lemma *brec-initial-abs*:

$\llbracket ls\text{-invar } \delta; hs\text{-invar } Qi \rrbracket$
 $\implies brec\text{-}\alpha \ (brec\text{-initial } Qi \ \delta) \in brw\text{-initial } (ls\text{-}\alpha \ \delta)$
 $\langle proof \rangle$

lemma *brec-pref-brw*:

assumes $RQRM[simp]: rqrm\text{-invar } rqrm \quad rqrm\text{-prop } (ls\text{-}\alpha \ \delta) \ rqrm$
assumes $INV[simp]: ls\text{-invar } \delta \quad hs\text{-invar } Qi$
shows $wa\text{-precise-refine } (det\text{-wa-wa } (brec\text{-det-algo } rqrm \ Qi \ \delta))$
 $(brw\text{-algo } (hs\text{-}\alpha \ Qi) \ (ls\text{-}\alpha \ \delta))$
 $brec\text{-}\alpha$
 $\langle proof \rangle$

lemma *brec-while-algo*:

assumes $RQRM[simp]: rqrm\text{-invar } rqrm \quad rqrm\text{-prop } (ls\text{-}\alpha \ \delta) \ rqrm$
assumes $INV[simp]: ls\text{-invar } \delta \quad hs\text{-invar } Qi$
shows $while\text{-algo } (det\text{-wa-wa } (brec\text{-det-algo } rqrm \ Qi \ \delta))$
 $\langle proof \rangle$

lemma *fst-brec- α* : $fst \ (brec\text{-}\alpha \ \Sigma) = hm\text{-}\alpha \ (fst \ \Sigma)$

$\langle proof \rangle$

lemmas *brec-invar-final* =

$wa\text{-precise-refine.transfer-correctness}[$
 $OF \ brec\text{-pref-brw } brw\text{-invar-final},$
 $unfolded \ fst\text{-brec-}\alpha]$

lemmas *brec-det-algo* = $det\text{-while-algo-intro}[OF \ brec\text{-while-algo}]$

definition *hta-is-empty-witness* $H ==$

$let \ rqrm = build\text{-rqrm } (hta\text{-}\delta \ H);$
 $(Q, -, -, qwit) = (while \ brec\text{-cond } (brec\text{-step } rqrm \ (hta\text{-}Qi \ H))$
 $(brec\text{-initial } (hta\text{-}Qi \ H) \ (hta\text{-}\delta \ H)))$

in

case $qwit$ of
 $None \Rightarrow None \ |$
 $Some \ q \Rightarrow (hm\text{-lookup } q \ Q)$

theorem (in *hashedTa*) *hta-is-empty-witness-correct*:
shows [*rule-format*]: *hta-is-empty-witness* $H = \text{Some } t$
 $\longrightarrow t \in \text{ta-lang } (hta-\alpha H)$ (is ?*T1*)
 $hta-is-empty-witness H = \text{None} \longrightarrow \text{ta-lang } (hta-\alpha H) = \{\}$ (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* :: - \Rightarrow *htai* \Rightarrow *bool*
where *htai-mem* == *hta-mem*
definition *htai-prod* :: *htai* \Rightarrow *htai* \Rightarrow *htai*
where *htai-prod* *H1 H2* == *hta-reindex* (*hta-prod* *H1 H2*)
definition *htai-prodWR* :: *htai* \Rightarrow *htai* \Rightarrow *htai*
where *htai-prodWR* *H1 H2* == *hta-reindex* (*hta-prodWR* *H1 H2*)
definition *htai-union* :: *htai* \Rightarrow *htai* \Rightarrow *htai*
where *htai-union* *H1 H2* == *hta-reindex* (*hta-union* *H1 H2*)
definition *htai-empty* :: *unit* \Rightarrow *htai*
where *htai-empty* == *hta-empty*
definition *htai-add-qi* :: - \Rightarrow *htai* \Rightarrow *htai*
where *htai-add-qi* == *hta-add-qi*
definition *htai-add-rule* :: - \Rightarrow *htai* \Rightarrow *htai*
where *htai-add-rule* == *hta-add-rule*
definition *htai-bwd-reduce* :: *htai* \Rightarrow *htai*
where *htai-bwd-reduce* == *hta-bwd-reduce*
definition *htai-is-empty-witness* :: *htai* \Rightarrow -
where *htai-is-empty-witness* == *hta-is-empty-witness*
definition *htai-ensure-idx-f* :: *htai* \Rightarrow *htai*
where *htai-ensure-idx-f* == *hta-ensure-idx-f*
definition *htai-ensure-idx-s* :: *htai* \Rightarrow *htai*
where *htai-ensure-idx-s* == *hta-ensure-idx-s*
definition *htai-ensure-idx-sf* :: *htai* \Rightarrow *htai*
where *htai-ensure-idx-sf* == *hta-ensure-idx-sf*

definition *htaip-prod* :: *htai* \Rightarrow *htai* \Rightarrow (*nat * nat,nat*) *hashedTa*
where *htaip-prod* == *hta-prod*
definition *htaip-prodWR* :: *htai* \Rightarrow *htai* \Rightarrow (*nat * nat,nat*) *hashedTa*
where *htaip-prodWR* == *hta-prodWR*
definition *htaip-reindex* :: (*nat * nat,nat*) *hashedTa* \Rightarrow *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: $hta-\alpha (hta-empty ()) = ta-empty$
 $hashedTa (hta-empty ())$

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 $hashedTa H \implies hta-\alpha (hta-add-qi qi H) = (ta-initial$
 $= insert qi (ta-initial (hta-\alpha H)), ta-rules = ta-rules (hta-\alpha H))$
 $hashedTa H \implies hashedTa (hta-add-qi qi H)$

Function: *hta-add-rule*

Adds a rule to the given automaton.

Relevant Lemmas

hashedTa.hta-add-rule-correct: $hashedTa H \implies hta-\alpha (hta-add-rule r H) =$
 $(ta-initial = ta-initial (hta-\alpha H), ta-rules = insert r (ta-rules (hta-\alpha$
 $H)))$
 $hashedTa H \implies hashedTa (hta-add-rule r H)$

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

hashedTa.hta-mem'-correct: $\llbracket \text{hashedTa } H; \text{hta-has-idx-f } H \rrbracket \implies \text{hta-mem}' t H = (t \in \text{ta-lang } (\text{hta-}\alpha H))$

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: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hta-}\alpha (\text{hta-prod } H1 H2) = \text{ta-fwd-reduce } (\text{ta-prod } (\text{hta-}\alpha H1) (\text{hta-}\alpha H2))$

$\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prod } H1 H2)$

hta-prod-correct: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{ta-lang } (\text{hta-}\alpha (\text{hta-prod } H1 H2)) = \text{ta-lang } (\text{hta-}\alpha H1) \cap \text{ta-lang } (\text{hta-}\alpha H2)$

$\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prod } H1 H2)$

hta-prod'-correct-aux: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2 \rrbracket \implies \text{hta-}\alpha (\text{hta-prod}' H1 H2) = \text{ta-fwd-reduce } (\text{ta-prod } (\text{hta-}\alpha H1) (\text{hta-}\alpha H2))$

$\llbracket \text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prod}' H1 H2)$

hta-prod'-correct: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2 \rrbracket \implies \text{ta-lang } (\text{hta-}\alpha (\text{hta-prod}' H1 H2)) = \text{ta-lang } (\text{hta-}\alpha H1) \cap \text{ta-lang } (\text{hta-}\alpha H2)$
 $\llbracket \text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prod}' H1 H2)$

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: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hta-}\alpha (\text{hta-prodWR } H1 H2) = \text{ta-prod } (\text{hta-}\alpha H1) (\text{hta-}\alpha H2)$
 $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prodWR } H1 H2)$

hta-prodWR-correct: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{ta-lang } (\text{hta-}\alpha (\text{hta-prodWR } H1 H2)) = \text{ta-lang } (\text{hta-}\alpha H1) \cap \text{ta-lang } (\text{hta-}\alpha H2)$
 $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-prodWR } H1 H2)$

Function: *hta-union*

Compute the union of two tree automata.

Relevant Lemmas

hta-union-correct': $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hta-}\alpha (\text{hta-union } H1 H2) = \text{ta-union-wrap } (\text{hta-}\alpha H1) (\text{hta-}\alpha H2)$
 $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-union } H1 H2)$

hta-union-correct: $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{ta-lang } (\text{hta-}\alpha (\text{hta-union } H1 H2)) = \text{ta-lang } (\text{hta-}\alpha H1) \cup \text{ta-lang } (\text{hta-}\alpha H2)$
 $\llbracket \text{hashedTa } H1; \text{hashedTa } H2 \rrbracket \implies \text{hashedTa } (\text{hta-union } H1 H2)$

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: $\llbracket \text{hashedTa } H; \text{hs.invar } Q \rrbracket \implies \text{hta-}\alpha (\text{hta-reduce } H \ Q) = \text{ta-reduce } (\text{hta-}\alpha \ H) (\text{hs.}\alpha \ Q)$
 $\llbracket \text{hashedTa } H; \text{hs.invar } Q \rrbracket \implies \text{hashedTa } (\text{hta-reduce } H \ Q)$

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

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

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: $\llbracket \text{hashedTa } H; \text{hta-is-empty-witness } H = \text{Some } t \rrbracket \implies t \in \text{ta-lang } (\text{hta-}\alpha \ H)$
 $\llbracket \text{hashedTa } H; \text{hta-is-empty-witness } H = \text{None} \rrbracket \implies \text{ta-lang } (\text{hta-}\alpha \ H) = \{\}$

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-force 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 were 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 developed 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 quietly 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 its 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 deficiencies² (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* correct³, 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.

²For 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.

³A simple example is the always-diverging function $f_{\text{div}} :: \text{bool} = \text{while } (\lambda x. \text{True}) \text{ id True}$ that is definable in HOL. The lemma $\forall x. x = \text{if } f_{\text{div}} \text{ then } x \text{ else } x$ 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 Universität Wien, 2009.