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 ......................................... 13  
      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 .................................. 24  
  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 .................................. 39
5 Executable Implementation of Tree Automata

5.1 Prelude ................................................. 41
  5.1.1 Ad-Hoc instantiations of generic Algorithms .......... 41
5.2 Generating Indices of Rules .............................. 42
5.3 Tree Automaton with Optional Indices ..................... 43
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 ............................................. 52
  5.6.1 Reindex Automaton .................................... 52
5.7 Union .................................................. 53
5.8 Operators to Construct Tree Automata ..................... 54
5.9 Backwards Reduction and Emptiness Check ................. 55
  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 ........................................ 69

6 Conclusion ................................................. 70

6.1 Efficiency of Generated Code ............................ 70
6.2 Future Work ............................................. 71
6.3 Trusted Code Base ....................................... 72
1 Introduction

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

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

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

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

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

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

1.1 Submission Structure

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

1.1.1 common/

This directory contains a collection of generally useful theories.

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

\(^1\)Currently only backward (utility) reduction is refined to executable code
1.1.2 common/bugfixes/

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

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

1.1.3 /

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

AbsAlgo.thy Algorithms on tree automata.

Ta_impl.thy Executable implementation of tree automata.

Ta.thy Formalization of tree automata and basic properties.

Tree.thy Formalization of trees.

document/ Contains files for latex document creation

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

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

TODO Todo list

1.1.4 code/

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

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

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

doTests.sh Shell-script to compile all test-cases and start the performance measurement. When 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 func-
tions 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

end

3 Tree Automata

theory Ta
imports Main ../Automatic-Refinement/Lib/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 \( q \rightarrow l q_1 \ldots q_n \), with the meaning that one can derive \( l(q_1 \ldots q_n) \) from the state \( q \).

```plaintext
datatype ('q,'l) ta-rule = RULE 'q 'l 'q list ( - → - )
```

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

— Rule deconstruction

```plaintext
fun lhs where lhs (q → l qs) = q
fun rhsq where rhsq (q → l qs) = qs
fun rhsl where rhsl (q → l qs) = l
— States in a rule
fun rule-states where rule-states (q → l qs) = insert q (set qs)
— States in a set of rules
```

```plaintext
definition δ-states δ == ∪ (rule-states 'δ)
— States in a tree automaton
```

```plaintext
definition ta-rstates TA = ta-initial TA ∪ δ-states (ta-rules TA)
— Symbols occurring in rules
```

```plaintext
definition δ-symbols δ == rhsl δ
— Nondeterministic, finite tree automaton (NFTA)
```

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

3.1.2 Acceptance

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

A tree is accepted in state \( q \), if it can be produced from \( q \) using the rules.
inductive accs :: ('Q', 'L) ta-rule set ⇒ 'L tree ⇒ 'Q ⇒ bool
where
  [ (q → f qs) ∈ δ; length ts = length qs;
     [∀ i. i < length qs ⇒ accs δ (ts ! i) (qs ! i)
    ] ⇒ accs δ (NODE f ts) q

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

lemma accs-laz: accs = accs-laz
  ⟨proof⟩

3.1.3 Language

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

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

3.2 Basic Properties

lemma rule-states-simp:
  rule-states x = (case x of (q → l qs) ⇒ insert q (set qs))
  ⟨proof⟩

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

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

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

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

lemma δ-statesI′: [(q → l qs) ∈ δ; qi ∈ set qs] ⇒ qi ∈ δ-states δ
  ⟨proof⟩
lemma \(\delta\)-states-accsI: \(\text{accs}\ \delta \ n \ q \Rightarrow q \in \delta\)-states \(\delta\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-states-union[simp]: \(\delta\)-states \((\delta \cup \delta')\) = \(\delta\)-states \(\delta\) \cup \(\delta\)-states \(\delta'\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-states-insert[simp]:
\(\delta\)-states \((\text{insert } r \ \delta)\) = \(\text{rule-states } r \cup \delta\)-states \(\delta)\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-states-mono: \([\delta \subseteq \delta'] \Rightarrow \delta\)-states \(\delta\) \subseteq \(\delta\)-states \(\delta'\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-states-finite[simp, intro]: finite \(\delta\) \Rightarrow finite \((\delta\)-states \(\delta)\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-statesE: \([q \in \delta\)-states \(\Delta)\):
\!!f qs. \([q \rightarrow f \ qs] \in \Delta \Rightarrow P; \allowbreak
!!q l f qs \ [. (ql \rightarrow f \ qs) \in \Delta; q \in \text{set } qs \ ] \Rightarrow P\]
\([\Rightarrow P\]
\(\langle proof\rangle\)

lemma \(\delta\)-symbolsI: \((q \rightarrow f \ qs) \in \delta \Rightarrow f \in \delta\)-symbols \(\delta\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-symbolsE:
assumes A: \(f \in \delta\)-symbols \(\delta\)
obtains q qs where \((q \rightarrow f \ qs) \in \delta\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-symbols-simps[simp]:
\(\delta\)-symbols \([\{\}] = \{\}
\(\delta\)-symbols \((\text{insert } r \ \delta) = \text{insert } (\text{rsl } r) \ (\delta\)-symbols \(\delta)\)
\(\delta\)-symbols \((\delta \cup \delta') = \delta\)-symbols \(\delta\) \cup \delta\)-symbols \(\delta'\)
\(\langle \text{proof} \rangle\)

lemma \(\delta\)-symbols-finite[simp, intro]:
finite \(\delta\) \Rightarrow finite \((\delta\)-symbols \(\delta)\)
\(\langle \text{proof} \rangle\)

lemma accs-mono: \([\text{accs } \delta \ n \ q; \delta \subseteq \delta'] \Rightarrow \text{accs } \delta' \ n \ q\)
\(\langle \text{proof} \rangle\)

context tree-automaton
begin
lemma initial-subset: ta-initial TA \subseteq ta-rstates TA
\(\langle \text{proof} \rangle\)
lemma states-subset: \(\delta\)-states \((\text{ta-rules } TA) \subseteq \text{ta-rstates } TA\)
\(\langle \text{proof} \rangle\)

10
lemma finite-states[simp, intro!]: finite (ta-rstates TA)
(proof)

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

lemmas is-subset = set-rev-mp[OF - initial-subset]
set-rev-mp[OF - states-subset]
end

3.3 Other Classes of Tree Automata

3.3.1 Automata over Ranked Alphabets

— All trees over ranked alphabet

inductive-set ranked-trees :: ('L ⇒ nat) ⇒ 'L tree set
for A where
\[ ∀ t ∈ set ts. t ∈ ranked-trees A; A f = Some (length ts) \]
⇒ NODE f ts ∈ ranked-trees A

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

context finite-alphabet
begin

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

lemma legal-rulesI:
[ r ∈ δ;
  rule-states r ⊆ Q;
  A (rhsl r) = Some (length (rhsq r))
] ⇒ r ∈ legal-rules Q
(proof)

lemma legal-rules-finite[simp, intro!]:
fixes Q :: 'Q set
assumes [simp, intro!]: finite Q
shows finite (legal-rules Q)
(proof)
end

11
locale ranked-tree-automaton =
    tree-automaton TA +
    finite-alphabet A
for TA :: ('Q,'L) tree-automaton-rec
and A :: 'L ⇒ nat +
assumes ranked: (q → f qs)∈δ ⇒ A f = Some (length qs)
begin

lemma rules-legal: r∈δ ⇒ r∈legal-rules Q
⟨proof⟩
lemma accs-is-ranked: accs δ t q ⇒ t∈ranked-trees A
⟨proof⟩
theorem lang-is-ranked: ta-lang TA ⊆ ranked-trees A
⟨proof⟩
end

3.3.2 Deterministic Tree Automata
— Deterministic, (bottom-up) finite tree automaton (DFTA)
locale det-tree-automaton = ranked-tree-automaton TA A
for TA :: ('Q,'L) tree-automaton-rec and A +
assumes deterministic: [ (q → f qs)∈δ; (q′ → f qs)∈δ ] ⇒ q=q'
begin
  theorem accs-unique: [ accs δ t q; accs δ t q' ] ⇒ q=q'
  ⟨proof⟩
end

3.3.3 Complete Tree Automata
locale complete-tree-automaton = det-tree-automaton TA A
for TA :: ('Q,'L) tree-automaton-rec and A +
assumes complete:
[ qs∈lists Q; A f = Some (length qs) ] ⇒ ∃ q. (q → f qs)∈δ
begin
  — In a complete DFTA, all trees can be labeled by some state
  theorem label-all: t∈ranked-trees A ⇒ ∃ q∈Q. accs δ t q
  ⟨proof⟩
end
3.4 Algorithms

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

3.4.1 Empty Automaton

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

theorem ta-empty-lang[simp]: ta-lang ta-empty = {} (proof)

theorem ta-empty-ta[simp, intro!]: tree-automaton ta-empty (proof)

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

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

3.4.2 Remapping of States

fun remap-rule where remap-rule f (q → l qs) = (f q) → l (map f qs)

definition ta-remap f TA == (\{\} ta-initial = f ' ta-initial TA, ta-rules = remap-rule f ' ta-rules TA)

lemma δ-states-remap[simp]: δ-states (remap-rule f ' δ) = f' δ-states δ (proof)

lemma remap-accs1: accs δ n q ⇒ accs (remap-rule f ' δ) n (f q) (proof)

lemma remap-lang1: t ∈ ta-lang TA ⇒ t ∈ ta-lang (ta-remap f TA) (proof)

lemma remap-accs2: [accs δ' n q']; δ'=(remap-rule f ' δ); q'=f q; inj-on f Q; q∈Q; δ-states δ ⊆ Q] ⇒ accs δ n q
lemma (in tree-automaton) remap-lang2:
assumes I: inj-on f (ta-rstates TA)
shows t∈ta-lang (ta-remap f TA) ⇒ t∈ta-lang TA
⟨proof⟩

theorem (in tree-automaton) remap-lang:
inj-on f (ta-rstates TA) ⇒ ta-lang (ta-remap f TA) = ta-lang TA
⟨proof⟩

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

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

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

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

3.4.3 Union

definition ta-union TA TA' ==
( ta-initial = ta-initial TA ∪ ta-initial TA',
  ta-rules = ta-rules TA ∪ ta-rules TA' )
⟩

— Given two disjoint sets of states, where no rule contains states from both sets, then any accepted tree is also accepted when only using one of the subsets of states and rules. This lemma and its corollaries capture the basic idea of the union-algorithm.

lemma accs-exclusive-aux:
[ accs δ n q; δn=δ∪δ'; δ-states δ ∩ δ-states δ' = {}; q∈δ-states δ ]
⇒ accs δ n q
⟨proof⟩

corollary accs-exclusive1:
[ accs (δ∪δ') n q; δ-states δ ∩ δ-states δ' = {}; q∈δ-states δ ]
⇒ accs δ n q

14
\textbf{corollary accs-exclusive2:}
\[ \text{accs}\ (\delta \cup \delta') n q; \delta\text{-states } \delta \cap \delta'\text{-states } \delta' = \{\}; \ q\in\delta'\text{-states } \delta' \] \[ \implies \text{accs}\ \delta' \ n \ q \]

\textbf{lemma ta-union-correct-aux1:}
fixes \( TA \ TA' \)
assumes \( TA: \text{tree-automaton} \ TA \)
assumes \( TA': \text{tree-automaton} \ TA' \)
assumes \( DJ: \text{ta-rstates } TA \cap \text{ta-rstates } TA' = \{\} \)
shows \( \text{ta-lang } (\text{ta-union } TA \ TA') = \text{ta-lang } TA \cup \text{ta-lang } TA' \)

\textbf{lemma ta-union-correct-aux2:}
fixes \( TA \ TA' \)
assumes \( TA: \text{tree-automaton} \ TA \)
assumes \( TA': \text{tree-automaton} \ TA' \)
shows \( \text{tree-automaton } (\text{ta-union } TA \ TA') \)

\textbf{theorem ta-union-correct:}
fixes \( TA \ TA' \)
assumes \( TA: \text{tree-automaton} \ TA \)
assumes \( TA': \text{tree-automaton} \ TA' \)
assumes \( DJ: \text{ta-rstates } TA \cap \text{ta-rstates } TA' = \{\} \)
shows \( \text{ta-lang } (\text{ta-union } TA \ TA') = \text{ta-lang } TA \cup \text{ta-lang } TA' \)
\text{tree-automaton } (\text{ta-union } TA \ TA') \)

\textbf{lemma ta-union-rta:}
fixes \( TA \ TA' \)
assumes \( TA: \text{ranked-tree-automaton} \ TA \)
assumes \( TA': \text{ranked-tree-automaton} \ TA' \)
shows \( \text{ranked-tree-automaton } (\text{ta-union } TA \ TA') \)

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

\textbf{datatype}\ ([q1,q2] usstate-wrapper = USW1 'q1 | USW2 'q2

\textbf{lemma usw-disjoint[simp]:}
USW1 \ X \cap USW2 \ Y = \{\} \remap-rule USW1 \ X \cap \remap-rule USW2 \ Y = \{\}
\textbf{lemma states-usw-disjoint[simp]:}
ta-rstates (ta-remap USW1 X) \cap ta-rstates (ta-remap USW2 Y) = \{\}
\textbf{proof}
**Lemma** \texttt{usw-inj-on}\{simp, intro\}:

- \texttt{inj-on USW1 X}
- \texttt{inj-on USW2 X}

\[\text{proof}\]

**Definition** \texttt{ta-union-wrap TA TA’} =

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

**Lemma** \texttt{ta-union-wrap-correct}:

- \texttt{fixes TA :: (‘Q1,’L) tree-automaton-rec}
- \texttt{fixes TA’ :: (‘Q2,’L) tree-automaton-rec}
- \texttt{assumes TA: tree-automaton TA}
- \texttt{assumes TA’: tree-automaton TA’}
- \texttt{shows ta-lang (ta-union-wrap TA TA’) = ta-lang TA \cup ta-lang TA’ (is ?T1)}
  \(\text{tree-automaton (ta-union-wrap TA TA’) (is ?T2)}\)

\[\text{proof}\]

**Lemma** \texttt{ta-union-wrap-rta}:

- \texttt{fixes TA TA’}
- \texttt{assumes TA: ranked-tree-automaton TA A}
- \texttt{assumes TA’: ranked-tree-automaton TA’ A}
- \texttt{shows ranked-tree-automaton (ta-union-wrap TA TA’) A}

\[\text{proof}\]

### 3.4.4 Reduction

**Definition** \texttt{reduce-rules} \(\delta P == \delta \cap \{ \ r. \ rule-states r \subseteq P \} \)

**Lemma** \texttt{reduce-rulesI}: \(\{ r \in \delta; \ rule-states r \subseteq P \} \implies r \in \text{reduce-rules} \delta P\)

\[\text{proof}\]

**Lemma** \texttt{reduce-rulesD}:

- \(\{ r \in \text{reduce-rules} \delta P \} \implies r \in \delta\)
- \(\{ r \in \text{reduce-rules} \delta P; q \in \text{rule-states} r \} \implies q \in P\)

\[\text{proof}\]

**Lemma** \texttt{reduce-rules-subset}: \texttt{reduce-rules} \(\delta P \subseteq \delta\)

\[\text{proof}\]

**Lemma** \texttt{reduce-rules-mono}: \(P \subseteq P’ \implies \text{reduce-rules} \delta P \subseteq \text{reduce-rules} \delta P’\)

\[\text{proof}\]

**Lemma** \texttt{δ-states-reduce-subset}:

- \texttt{shows δ-states (reduce-rules δ Q) \subseteq δ-states δ \cap Q}\)

\[\text{proof}\]

**Lemmas** \texttt{δ-states-reduce-subsetI} = \texttt{set-rev-mp[OF - δ-states-reduce-subset]}
definition ta-reduce
\[
:: (\ ('Q',L) \ \text{tree-automaton-rec} \Rightarrow ('Q \ \text{set}) \Rightarrow ('Q,L) \ \text{tree-automaton-rec})
\]

where ta-reduce TA P ==
\[
\begin{align*}
&\emptyset \text{ ta-initial } = \text{ta-initial} \ \text{TA} \cap P, \\
&\text{ta-rules } = \text{reduce-rules} (\text{ta-rules} \ \text{TA}) \ P \\
\end{align*}
\]

— Reducing a tree automaton preserves the tree automata invariants

theorem ta-reduce-inv: assumes A: tree-automaton TA
shows tree-automaton (ta-reduce TA P)

⟨proof⟩

lemma reduce-\(\delta\)-states-rules\(\text{[simp]}\): 
\[
(\text{ta-rules} (\text{ta-reduce} \ \text{TA} (\delta\text{-states} (\text{ta-rules} \ \text{TA})))) = \text{ta-rules} \ \text{TA}
\]

⟨proof⟩

lemma ta-reduce-\(\delta\)-states:
\[
\text{ta-lang} (\text{ta-reduce} \ \text{TA} (\delta\text{-states} (\text{ta-rules} \ \text{TA}))) = \text{ta-lang} \ \text{TA}
\]

⟨proof⟩

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

— Forward successors

inductive-set \(f\text{-succ} \text{ for } \delta\) where
\[
[(q \rightarrow l qs)\in\delta; q'\in\text{set qs}] \implies (q,q') \in f\text{-succ} \ \delta
\]

— Alternative characterization of forward successors

lemma \(f\text{-succ-alt} : f\text{-succ} \ \delta \ = \ \{(q,q') \exists l qs. (q \rightarrow l qs)\in\delta \land q'\in\text{set qs}\}
\]

⟨proof⟩

definition \(f\text{-accessible} \ \delta \ Q0 \ == \ (f\text{-succ} \ \delta)^*'' \ Q0
\]

— Alternative characterization of forward accessible states. The initial states are
forward accessible, and if there is a rule whose lhs-state is forward-accessible,
all rhs-states of that rule are forward-accessible, too.

inductive-set \(f\text{-accessible-alt} :: (\ ('Q',L) \ \text{ta-rule set} \Rightarrow 'Q \ \text{set} \Rightarrow 'Q \ \text{set}) \ for \ \delta \ Q0\)

where
\[
\begin{align*}
&\text{fa-refl: } q0\in Q0 \implies q0 \in f\text{-accessible-alt} \ \delta \ Q0 \ |\\
&\text{fa-step: } [ q\in f\text{-accessible-alt} \ \delta \ Q0; (q \rightarrow l qs)\in\delta; q'\in\text{set qs} ] \\
&\implies q' \in f\text{-accessible-alt} \ \delta \ Q0
\end{align*}
\]

lemma \(f\text{-accessible-alt} : f\text{-accessible} \ \delta \ Q0 = f\text{-accessible-alt} \ \delta \ Q0
\]

⟨proof⟩

lemmas \(f\text{-accessibleI} = f\text{-accessible-alt} \text{.intros}[\text{folded } f\text{-accessible-alt}]
\]

lemmas \(f\text{-accessibleE} = f\text{-accessible-alt} \text{.cases}[\text{folded } f\text{-accessible-alt}]
\]
**lemma** \( f\text{-succ-finite}[\text{simp}, \text{intro}]: \) finite \( \delta \implies \) finite \( f\text{-succ} \delta \)

\langle proof \rangle

**lemma** \( f\text{-accessible-mono}: Q \subseteq Q' \implies x \in f\text{-accessible} \delta \ Q \implies x \in f\text{-accessible} \delta \ Q' \)

\langle proof \rangle

**lemma** \( f\text{-accessible-prepend}:
\[
\left[ (q \rightarrow l\ qs) \in \delta; \ q' \in \text{set}\ qs; \ x \in f\text{-accessible} \delta \ \{q'\} \right] \\
\implies x \in f\text{-accessible} \delta \ \{q\}
\]

\langle proof \rangle

**lemma** \( f\text{-accessible-subset}: q \in f\text{-accessible} \delta \ Q \implies q \in Q \cup \delta\text{-states} \delta \)

\langle proof \rangle

**lemma** \( \text{(in tree-automaton) f-accessible-in-states}:
\[
q \in f\text{-accessible} \ (\text{ta-rules} TA) (\text{ta-initial} TA) \implies q \in \text{ta-rstates} TA
\]

\langle proof \rangle

**lemma** \( f\text{-accessible-refl-inter-simp}[\text{simp}]: Q \cap f\text{-accessible} r Q = Q \)

\langle proof \rangle

**lemma** \( \text{accs-reduce-f-acc}:[\text{simp}]: \text{accs} \delta t q \implies \text{accs} (\text{reduce-rules} \ \delta (f\text{-accessible} \delta \ \{q\})) t q \)

\langle proof \rangle

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

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

**theorem** \( \text{ta-reduce-f-acc}[\text{simp}]: \text{ta-lang} (\text{ta-fwd-reduce} TA) = \text{ta-lang} TA \)

\langle proof \rangle

---

**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** \( b\text{-accessible} :: (\ 'Q, \ 'L) \text{ ta-rule set} \Rightarrow 'Q \text{ set} \)

**for** \( \delta \)

**where**
\[
\left[ (q \rightarrow l\ qs) \in \delta; \ (!!x. x \in \text{set}\ qs \implies x \in b\text{-accessible} \delta \right] \implies q \in b\text{-accessible} \delta
\]

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

\langle proof \rangle

**lemma** \( \text{accs-is-b-accessible}: \text{accs} \delta t q \implies q \in b\text{-accessible} \delta \)

\langle proof \rangle
lemma b-acc-subset-\(\delta\)-states\(\delta\): \(x \in \text{b-accessible } \delta \implies x \in \delta\text{-states } \delta\)
(proof)

lemma b-acc-subset-\(\delta\)-states: b-accessible \(\delta\) \(\subseteq\) \(\delta\)-states \(\delta\)
(proof)

lemma b-acc-finite\(\text{[simp, intro]}\): finite \(\delta\) \(\implies\) finite (b-accessible \(\delta\))
(proof)

lemma b-accessible-is-accs:
\[ \text{finite } \delta = \implies \text{finite } (\text{b-accessible } \delta) \]
(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\(\text{[simp]}\): ta-lang (ta-bwd-reduce TA) = ta-lang TA
(proof)

theorem empty-if-no-b-accessible:
\[ \text{ta-lang } TA = \{\} \iff \text{ta-initial } TA \cap \text{b-accessible } (\text{ta-rules } TA) = \{\} \]
(proof)

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

— Product rule

fun r-prod where
\[ r\text{-prod } (q_1 \rightarrow l q_{s1}) (q_2 \rightarrow l q_{s2}) = ((q_1, q_2) \rightarrow l l (\text{zip } q_{s1} q_{s2})) \]
— Product rules
definition \(\delta\)-prod \(\delta_1\) \(\delta_2\) \(==\) 
\[ r\text{-prod } (q_1 \rightarrow l q_{s1}) (q_2 \rightarrow l q_{s2}) \mid q_1 q_2 l q_{s1} q_{s2}. \]
\[ \text{length } q_{s1} = \text{length } q_{s2} \wedge \]
\[ (q_1 \rightarrow l q_{s1}) \in \delta_1 \wedge \]
\[ (q_2 \rightarrow l q_{s2}) \in \delta_2 \]

lemma \(\delta\)-prodI:
\[ \text{length } q_{s1} = \text{length } q_{s2}; \]
\[ (q_1 \rightarrow l q_{s1}) \in \delta_1; \]
\[ (q_2 \rightarrow l q_{s2}) \in \delta_2 \implies ((q_1, q_2) \rightarrow l (\text{zip } q_{s1} q_{s2})) \in \delta\text{-prod } \delta_1 \delta_2 \]
(proof)

lemma \(\delta\)-prodE:

\begin{align*}
\mathcal{r} &\in \delta \cdot \prod \delta_1 \delta_2; \\
\langle [\text{length } qs_1 = \text{length } qs_2; \quad (q_1 \rightarrow l qs_1) \in \delta_1; \quad (q_2 \rightarrow l qs_2) \in \delta_2; \\
r = ((q_1, q_2) \rightarrow l (\text{zip } qs_1 qs_2)) \rangle \Rightarrow P
\end{align*}

\text{lemma } \delta \cdot \prod \text{-sound:}
\begin{proof}
assumes A: accs (\delta \cdot \prod \delta_1 \delta_2) t (q_1, q_2)
shows accs \delta_1 t q_1 \quad accs \delta_2 t q_2
\end{proof}

\text{lemma } \delta \cdot \prod \text{-precise:}
\begin{proof}
\langle [\text{accs } \delta_1 t q_1; \text{accs } \delta_2 t q_2 ] \Rightarrow \text{accs } (\delta \cdot \prod \delta_1 \delta_2) t (q_1, q_2)
\end{proof}

\text{lemma } \delta \cdot \prod \text{-empty[simp]:}
\begin{proof}
\delta \cdot \prod \{\} = \{\}
\end{proof}

\text{lemma } \delta \cdot \prod \text{-2sng[simp]:}
\begin{proof}
\langle [\text{rhsl } r_1 \neq \text{rhsl } r_2 ] \Rightarrow \delta \cdot \prod \{r_1\} \{r_2\} = \{\}
\rangle \\\langle [\text{length } (\text{rhsq } r_1) \neq \text{length } (\text{rhsq } r_2) ] \Rightarrow \delta \cdot \prod \{r_1\} \{r_2\} = \{\}
\rangle \\\langle [\text{rhsl } r_1 = \text{rhsl } r_2; \text{length } (\text{rhsq } r_1) = \text{length } (\text{rhsq } r_2) ] \\
\Rightarrow \delta \cdot \prod \{r_1\} \{r_2\} = \{r \cdot \prod r_1 r_2\}
\end{proof}

\text{lemma } \delta \cdot \prod \text{-Un[simp]:}
\begin{proof}
\delta \cdot \prod (\delta \cdot \prod \delta_1 \delta_2) \cup \delta \cdot \prod \delta_1 \delta_2' = \delta \cdot \prod \delta_1 \delta_2 \cup \delta \cdot \prod \delta_1 \delta_2'
\end{proof}

The next two definitions are solely for technical reasons. They are required to allow simplification of expressions of the form \(\delta \cdot \prod \cdot (\text{insert } r \delta_1) \delta_2\) or \(\delta \cdot \prod \cdot \delta_1 (\text{insert } r \delta_2)\), without making the simplifier loop.

\text{definition } \delta \cdot \prod \text{-sng1 } r \delta_2 ==
\begin{proof}
case r of (q_1 \rightarrow l qs_1) \Rightarrow \\
\{ r \cdot \prod r (q_2 \rightarrow l qs_2) | \quad q_2 qs_2, \text{length } qs_1 = \text{length } qs_2 \land (q_2 \rightarrow l qs_2) \in \delta_2 \\
\}
\end{proof}

\text{definition } \delta \cdot \prod \text{-sng2 } \delta_1 r ==
\begin{proof}
case r of (q_2 \rightarrow l qs_2) \Rightarrow \\
\{ r \cdot \prod r (q_1 \rightarrow l qs_1) r | \quad q_1 qs_1, \text{length } qs_1 = \text{length } qs_2 \land (q_1 \rightarrow l qs_1) \in \delta_1 \\
\}
\end{proof}

\text{lemma } \delta \cdot \prod \text{-sng-alt:}
\begin{proof}
\delta \cdot \prod \text{-sng1 } r \delta_2 = \delta \cdot \prod \{r\} \delta_2
\end{proof}
\[ \delta_{-prod-sng2} r = \delta_{-prod} \delta_1 \{r\} \]

\[\langle \text{proof} \rangle\]

**lemmas** \(\delta_{-prod-insert} = \)
\(\delta_{-prod-Un}(1)\)\[where \ ? \delta_1.0=\{x\}, \ simplified, \ folded \ \delta_{-prod-sng-alt}\]
\(\delta_{-prod-Un}(2)\)\[where \ ? \delta_2.0=\{x\}, \ simplified, \ folded \ \delta_{-prod-sng-alt}\]

for \(x\)

---

**Product automaton**

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

**lemma** \(\text{ta-prod-correct-aux1}:\)
\(\text{ta-lang} (\text{ta-prod} TA1 \ TA2) = \text{ta-lang} TA1 \cap \text{ta-lang} TA2\)

\[\langle \text{proof} \rangle\]

**lemma** \(\delta_{-states-cart}:\)
\(q \in \delta_{-states} (\delta_{-prod} \delta_1 \delta_2) \implies q \in \delta_{-states} \delta_1 \times \delta_{-states} \delta_2\)

\[\langle \text{proof} \rangle\]

**lemma** \(\delta_{-prod-finite} \ [\text{simp, intro}]:\)
\(\text{finite} \ \delta_1 \implies \text{finite} \ \delta_2 \implies \text{finite} (\delta_{-prod} \delta_1 \delta_2)\)

\[\langle \text{proof} \rangle\]

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

\[\langle \text{proof} \rangle\]

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

\[\langle \text{proof} \rangle\]

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

\[\langle \text{proof} \rangle\]

### 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-tree-automaton

begin

— Left-hand side of subset rule for given symbol and rhs
definition δ ss-lhs f ss ==
\{ \ q \ | \ q \ qs. (q \rightarrow f qs) \in \delta \wedge \text{list-all-zip} \ (op \in) \ qs \ ss \ } 

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

lemma δ ss I:
assumes A: A f = Some (length ss)
ss \in \text{lists} \{ \ s. \ s \subseteq \text{ta-rstates TA} \}
shows ( (δ ss-lhs f ss) \rightarrow f ss ) \in \delta ss
⟨proof⟩

lemma δ ss-subset[simp, intro!]: δ ss-lhs f ss \subseteq Q
⟨proof⟩

lemma δ ss-finite[simp, intro!]: finite δ ss
⟨proof⟩

lemma δ ss-det: \[ \left[ (q \rightarrow f qs) \in \delta ss; (q' \rightarrow f qs) \in \delta ss \right] \implies q=q' \]
⟨proof⟩

lemma δ ss-accs-sound:
assumes A: accs δ t q
obtains s where
s \subseteq Q
q \in s
accs δ ss t s
⟨proof⟩

lemma δ ss-accs-precise:
assumes A: accs δ ss t s \ q \in s
shows accs δ t q
⟨proof⟩

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

theorem detTA-is-ta[simp, intro]:
det-tree-automaton detTA A
⟨proof⟩
Theorem detTA-lang[simp]:
\[ \text{ta-lang (detTA)} = \text{ta-lang TA} \]
(\text{proof})

Theorems detTA-correct = detTA-is-ta detTA-lang
end

3.4.7 Completion

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

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

lemma Qcomplete-finite[simp, intro!]: finite Qcomplete
(\text{proof})
definition δcomplete :: ('Q option, 'L) ta-rule set where
\[ \delta_\text{complete} == (\text{remap-rule Some } \delta) \cup \{ (\text{None }\rightarrow f \text{ qs}) \mid f \text{ qs. } \]
\[ A f = \text{Some (length qs)} \]
\[ \land \text{qs} \in \text{lists Qcomplete} \]
\[ \land \neg (\exists qo qso. (qo }\rightarrow f \text{ qso}) \in \delta \land \text{qs} = \text{map Some qso } \}

lemma δ-states-complete: q ∈ δ-states δcomplete ⇒ q ∈ Qcomplete
(\text{proof})

definition completeTA == ( ta-initial = Some'Qi, ta-rules = δcomplete )

lemma δcomplete-finite[simp, intro]: finite δcomplete
(\text{proof})

Theorem completeTA-is-ta: complete-tree-automaton completeTA A
(\text{proof})

Theorem completeTA-lang: ta-lang completeTA = ta-lang TA
(\text{proof})

Theorems completeTA-correct = completeTA-is-ta completeTA-lang
end
3.4.8 Complement

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

context complete-tree-automaton

begin

— Complement automaton, i.e. that accepts exactly the trees not accepted by this automaton

definition complementTA == ()
  ta-initial = Q - Qi,
  ta-rules = δ

lemma cta-rules[simp]; ta-rules complementTA = δ
(proof)

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

end

3.5 Regular Tree Languages

3.5.1 Definitions

— Regular languages over alphabet \( A \)

definition regular-languages :: ('L → nat) ⇒ 'L tree set set
where regular-languages A ==
  { ta-lang TA | (TA::(nat,'L) tree-automaton-rec).
    ranked-tree-automaton TA A }

lemma rtlE:
  fixes L :: 'L tree set
  assumes A: L∈regular-languages A
  obtains TA::(nat,'L) tree-automaton-rec where
    L=ta-lang TA
    ranked-tree-automaton TA A
(proof)

context ranked-tree-automaton
begin

lemma (in ranked-tree-automaton) rtlI[simp]:
  shows ta-lang TA ∈ regular-languages A
It is sometimes more handy to obtain a complete, deterministic tree automaton accepting a given regular language.

\textbf{theorem} obtain-complete:
\begin{itemize}
\item obtains $TAC::(Q\text{ set option},'L)\text{ tree-automaton-rec where}$
\item $ta-lang\ TAC = ta-lang\ TA$
\item complete-tree-automaton $TAC\ A$
\end{itemize}
\textbf{end}

\textbf{lemma} rtlE-complete:
\begin{itemize}
\item fixes $L::'L\text{ tree set}$
\item assumes $A: L\in\text{regular-languages}\ A$
\item obtains $TA::(nat,'L)\text{ tree-automaton-rec where}$
\item $L=ta-lang\ TA$
\item complete-tree-automaton $TA\ A$
\end{itemize}
\textbf{end}

\textbf{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.

\textbf{theorem (in finite-alphabet)} rtl-empty\[\text{simp, intro}!]: \{\} \in\text{regular-languages}\ A$
\textbf{end}

\textbf{theorem} rtl-union-closed:
\begin{itemize}
\item $L1\in\text{regular-languages}\ A; L2\in\text{regular-languages}\ A$
\item $L1\cup L2 \in\text{regular-languages}\ A$
\end{itemize}
\textbf{end}

\textbf{theorem} rtl-inter-closed:
\begin{itemize}
\item $L1\in\text{regular-languages}\ A; L2\in\text{regular-languages}\ A$
\item $L1\cap L2 \in\text{regular-languages}\ A$
\end{itemize}
\textbf{end}

\textbf{theorem} rtl-complement-closed:
\begin{itemize}
\item $L\in\text{regular-languages}\ A \implies \text{ranked-trees}\ A - L \in\text{regular-languages}\ A$
\end{itemize}
\textbf{end}

\textbf{theorem (in finite-alphabet)} rtl-univ:
\begin{itemize}
\item ranked-trees $A \in\text{regular-languages}\ A$
\end{itemize}
\textbf{end}
proof

theorem rtl-diff-closed:
  fixes L1 :: 'L tree set
  assumes A[simp]: L1 ∈ regular-languages A  L2∈regular-languages A
  shows L1−L2 ∈ regular-languages A
(proof)

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

end

4 Abstract Tree Automata Algorithms

theory AbsAlgo
imports Ta 
../Collections/Examples/ICF/Exploration 
../Collections/ICF/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. 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

```latex
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)

$(\text{map (faccs δ) ts = map (λt. {q . accs δ t q}) ts})$ (is ?T2)

⟨proof⟩

**Theorem** faccs-correct1: $q∈faccs δ n ⇒ accs δ n q$

⟨proof⟩

**Theorem** faccs-correct2: $accs δ n q ⇒ q∈faccs δ n$

⟨proof⟩

**Theorems** faccs-correct = faccs-correct1 faccs-correct2

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

### 4.2 Backward Reduction and Emptiness Check

#### 4.2.1 Auxiliary Definitions

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

**Inductive-set** bacc-step :: (′Q,′L) ta-rule set ⇒ ′Q set ⇒ ′Q set

for δ Q

where

$[\ r∈δ; \ set (rhsq r) ⊆ Q \ ] \ ⇒ \ lhs r ∈ bacc-step δ Q$

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

**Lemma** b-accs-as-closed:

assumes $A: \ bacc-step δ Q ⊆ Q$

shows $b-accessible δ ⊆ Q$

⟨proof⟩

#### 4.2.2 Algorithms

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

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

α On this level, the state consists of a set of discovered states and a workset.

α’ On this level, the state consists of a set of discovered states, a workset and a map from rules to number of undiscovered rhs states. This map can be used to make the discovery of rules that have to be considered more efficient.

α - Level: — A state contains the set of discovered states and a workset type-synonym (′Q,′L) br-state = ′Q set × ′Q set — Set of states that are non-empty (accept a tree) after adding the state q to the set of discovered states definition br-dsq :: (′Q,′L) ta-rule set ⇒ ′Q ⇒ (′Q,′L) br-state ⇒ ′Q set where br-dsq δ q == λ(Q,W). { lhs r | r. r∈δ ∧ set (rhsq r) ⊆ (Q−(W−{q})) } — Description of a step: One state is removed from the workset, and all new states that become non-empty due to this state are added to, both, the workset and the set of discovered states inductive-set br-step :: (′Q,′L) ta-rule set ⇒ ((′Q,′L) br-state × (′Q,′L) br-state) set for δ where [ q∈W; Q’ = Q ∪ br-dsq δ q (Q,W); W’ = W − {q} ∪ (br-dsq δ q (Q,W) − Q) ] ⇒ ((Q,W),(Q’,W’))∈br-step δ — Termination condition for backwards reduction: The workset is empty definition br-cond :: (′Q,′L) br-state set where br-cond δ == ((Q,W). W≠{}) — Termination condition for emptiness check: The workset is empty or a non-empty initial state has been discovered definition bre-cond Qi :: (′Q,′L) br-state set where bre-cond Qi == {((Q,W). W≠{}) ∧ (Qi∩Q={})} — Set of all states that occur on the lhs of a constant-rule definition br-iq :: (′Q,′L) ta-rule set ⇒ ′Q set where br-iq δ == { lhs r | r. r∈δ ∧ rhsq r = [] } — Initial state for the iteration definition br-initial :: (′Q,′L) ta-rule set ⇒ (′Q,′L) br-state where br-initial δ == (br-iq δ, br-iq δ)
— Invariant for the iteration:
  • States on the workset have been discovered
  • Only accessible states have been discovered
  • If a state is non-empty due to a rule whose rhs-states have been discovered and processed (i.e. are in \( Q - W \)), then the lhs state of the rule has also been discovered.
  • The set of discovered states is finite

\[
\text{definition } \text{br-invar} :: \langle Q, L \rangle \text{-rule set } \Rightarrow \langle Q', L \rangle \text{ br-state set}
\]
\[
\text{where } \text{br-invar } \delta == \{(Q, W) \}
\]
\[
W \subseteq Q \land Q \subseteq b\text{-accessible } \delta \land \\
bacc-step \delta (Q - W) \subseteq Q \land \\
\text{finite } Q
\]

\[
\text{definition } \text{br-algo } \delta == ()
\]
\[
\text{wa-cond} = \text{br-cond},
\]
\[
\text{wa-step} = \text{br-step } \delta,
\]
\[
\text{wa-initial} = \{\text{br-initial } \delta\},
\]
\[
\text{wa-invar} = \text{br-invar } \delta
\]
\[
()
\]

\[
\text{definition } \text{bre-algo } Q_i \delta == ()
\]
\[
\text{wa-cond} = \text{bre-cond } Q_i,
\]
\[
\text{wa-step} = \text{br-step } \delta,
\]
\[
\text{wa-initial} = \{\text{br-initial } \delta\},
\]
\[
\text{wa-invar} = \text{br-invar } \delta
\]
\[
()
\]

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

\[
\text{definition } \text{br-termrel } \delta == \\
\{(Q', Q). Q \subset Q' \land Q' \subseteq \text{b-accessible } \delta\} \langle*\text{lex*}⟩ \text{ finite-psubset}
\]

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

\[
\text{lemma } \text{br-termrel-uf}[\text{simp, intro}!]: \text{finite } \delta \Rightarrow \text{wf } \langle\text{br-termrel } \delta\rangle
\]
\[
\langle\text{proof}\rangle
\]

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

\[
\text{lemma } \text{br-step-in-termrel}: \langle\text{proof}\rangle
\]
\[
\text{assumes } A: \Sigma \in \text{br-cond} \quad \Sigma \in \text{br-invar } \delta \quad \langle\Sigma, \Sigma'\rangle \in \text{br-step } \delta
\]
\[
29
\]
\( \Sigma' \), \((\Sigma', \Sigma) \in \text{br-termrel } \delta \)

(proof)

**lemma** \( \text{br-invar-initial} \): \( \text{finite } \delta \implies (\text{br-initial } \delta) \in \text{br-invar } \delta \)

(proof)

**lemma** \( \text{br-invar-step} \):

assumes [simp]: \( \text{finite } \delta \)

assumes \( \Lambda: \Sigma \in \text{br-cond } \Sigma \in \text{br-invar } \delta \) \((\Sigma, \Sigma') \in \text{br-step } \delta \)

shows \( \Sigma' \in \text{br-invar } \delta \)

(proof)

**lemma** \( \text{br-invar-final} \):

\( \forall \Sigma. \Sigma \in \text{wa-invar } (\text{br-algo } \delta) \land \Sigma \notin \text{wa-cond } (\text{br-algo } \delta) \)

\( \implies \text{fst } \Sigma = b\text{-accessible } \delta \)

(proof)

**theorem** \( \text{br-while-algo} \):

assumes \( \text{FIN} \) [simp]: \( \text{finite } \delta \)

shows \( \text{while-algo } (\text{br-algo } \delta) \)

(proof)

**lemma** \( \text{bre-invar-final} \):

\( \forall \Sigma. \Sigma \in \text{wa-invar } (\text{bre-algo } Q_i \delta) \land \Sigma \notin \text{wa-cond } (\text{bre-algo } Q_i \delta) \)

\( \implies (\text{fst } \Sigma = b\text{-accessible } \delta) \leftrightarrow (Q_i \cap \text{fst } \Sigma = \{\}) \)

(proof)

**theorem** \( \text{bre-while-algo} \):

assumes \( \text{FIN} \) [simp]: \( \text{finite } \delta \)

shows \( \text{while-algo } (\text{bre-algo } Q_i \delta) \)

(proof)

\( \alpha' \)-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** \( (Q', L) \, \text{br'-state} = \{'Q set \times 'Q set \times (('Q',L) ta-rule } \rightarrow \text{nat} \)

--- Abstraction to \( \alpha \)-level

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

where \( \text{br'-} \alpha = \lambda(Q,W,rcm). (Q,W)) \)
definition \( br^-\text{invar-add} :: (Q',L) \) ta-rule set \( \Rightarrow (Q',L) \) \( br^-\text{state} \) set  
where \( br^-\text{invar-add} \ \delta \ = \ ((Q,W,rcm), \forall r \in \delta. \ \text{rcm} \ r \ = \ \text{Some} \ (\text{card} \ (\text{set} \ (\text{rhsq} \ r)) \ - \ (Q \ - \ W))) \ \land \ 
\{ \text{lhs} \ r \ | \ r \in \delta \ \land \ \text{the} \ (\text{rcm} \ r) \ = \ \emptyset \} \subseteq Q \} \)

definition \( br^-\text{invar} :: (Q',L) \) ta-rule set \( \Rightarrow (Q',L) \) \( br^-\text{state} \) set  
where \( br^-\text{invar} \ \delta \ = \ ((Q,W,rcm), \forall r \in \delta. \ \text{rcm} \ r \ = \ \text{Some} \ (\text{card} \ (\text{set} \ (\text{rhsq} \ r)) \ - \ (Q \ - \ W))) \ \land \ 
\{ \text{lhs} \ r \ | \ r \in \delta \ \land \ \text{the} \ (\text{rcm} \ r) \ = \ \emptyset \} \subseteq Q \} \)

inductive-set \( \text{br}^-\text{step} \)  
:: (Q',L) ta-rule set \( \Rightarrow ((Q',L) \ \text{br}^-\text{state} \times (Q',L) \ \text{br}^-\text{state}) \) set  
for \( \delta \) where  
\[
[ \ q \in W; \ Q' = Q \ \cup \ \{ \text{lhs} \ r \ | \ r \in \delta \ \land \ q \in \text{set} \ (\text{rhsq} \ r) \ \land \ \text{the} \ (\text{rcm} \ r) \leq 1 \}; \ W' = (W \ - \ \{ q \}) \ 
\cup \ (\{ \text{lhs} \ r \ | \ r \in \delta \ \land \ q \in \text{set} \ (\text{rhsq} \ r) \ \land \ \text{the} \ (\text{rcm} \ r) \leq 1 \} \ 
\ - \ Q); \ 
\ !r. \ r \in \delta \ \Rightarrow \ \text{rcm} \ r \ = \ ( \text{if} \ q \in \text{set} \ (\text{rhsq} \ r) \ \text{then} \ 
\ \text{Some} \ \text{the} \ (\text{rcm} \ r) \ - \ 1) \ 
\ \text{else} \ \text{rcm} \ r \ ) \ ] \ \Rightarrow \ ((Q,W,rcm),(Q',W',rcm')) \in \text{br}^-\text{step} \ \delta \]

definition \( \text{br}^-\text{cond} :: (Q',L) \) \( \text{br}^-\text{state} \) set  
where \( \text{br}^-\text{cond} \ = \ ((Q,W,rcm), \ W \not= \{\}) \)

definition \( \text{bre}^-\text{cond} :: Q \) set \( \Rightarrow (Q',L) \) \( \text{br}^-\text{state} \) set  
where \( \text{bre}^-\text{cond} \ Q_\iota \ = \ ((Q,W,rcm), \ W \not= \{\}) \land (Q_\iota \cap Q = \{\}) \)

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

definition \( \text{br}^-\text{algo} \ \delta \ = \ \emptyset \)  
\( \text{wa-cond} = \text{br}^-\text{cond}, \)  
\( \text{wa-step} = \text{br}^-\text{step} \ \delta, \)  
\( \text{wa-initial} = \text{br}^-\text{initial} \ \delta, \)  
\( \text{wa-invar} = \text{br}^-\text{invar} \ \delta \)

definition \( \text{bre}^-\text{algo} \ Q_\iota \ \delta \ = \ \emptyset \)  
\( \text{wa-cond} = \text{bre}^-\text{cond} \ Q_\iota, \)  
\( \text{wa-step} = \text{br}^-\text{step} \ \delta, \)  
\( \text{wa-initial} = \text{br}^-\text{initial} \ \delta, \)  
\( \text{wa-invar} = \text{br}^-\text{invar} \ \delta \)

lemma \( \text{br}^-\text{step-invar}: \)
assumes finite[simp]: finite δ
assumes INV: Σ∈br'\text{-}invar\text{-}add δ \quad br'\text{-}\alpha \Sigma \in br\text{-}invar δ
assumes STEP: (\Sigma,\Sigma') \in br'\text{-}step δ
shows Σ'\in br'\text{-}invar\text{-}add δ
(proof)

lemma br'\text{-}invar\text{-}initial:
\quad br'\text{-}initial δ \subseteq \mathit{br'}\text{-}invar\text{-}add δ
(proof)

lemma br'\text{-}rcm\text{-}aux\text{'}:
\quad \begin{array}{l}
\quad \text{assumes A: (Q,W,rcm) ∈ br'\text{-}invar δ q∈W} \\
\quad \text{shows } \{lhs r | r ∈ δ \land q ∈ set (rhsq r) \land \text{the (rcm r)} ≤ Suc 0\}
\quad = \{lhs r | r ∈ δ \land q ∈ set (rhsq r) \land set (rhsq r) ⊆ (Q - (W - \{q\}))\}
\quad \end{array}
(proof)

lemma br'\text{-}invar\text{-}QcD:
\quad \begin{array}{l}
\quad \text{(Q,W,rcm) ∈ br'\text{-}invar δ} \implies \{lhs r | r ∈ δ \land set (rhsq r) \subseteq Q \}
\quad \subseteq Q
\quad \end{array}
(proof)

lemma br'\text{-}rcm\text{-}aux2:
\quad \begin{array}{l}
\quad \text{assumes A: (Q,W,rcm) ∈ br'\text{-}invar δ q∈W} \\
\quad \text{shows } Q \cup \text{br-dsq δ q (Q,W)}
\quad = Q \cup \{lhs r | r ∈ δ \land q ∈ set (rhsq r) \land \text{the (rcm r)} ≤ Suc 0\}
\quad \end{array}
(proof)

lemma 32
lemma $br'$-cond-abs: $\Sigma \in br'$-cond $\leftrightarrow (br' - \alpha \Sigma) \in br$-cond
(proof)

lemma bre'$-cond-abs: $\Sigma \in bre'$-cond $Qi \leftrightarrow (br' - \alpha \Sigma) \in bre$-cond $Qi$
(proof)

lemma $br'$-invar-abs: $br'$-invar $\delta \subseteq br$-invar $\delta$
(proof)

theorem $br'$-pref-br: wa-precise-refine $(br'$-algo $\delta)$ $(br$-algo $\delta)$ $br'$-\alpha
(proof)

interpretation $br'$-pref: wa-precise-refine $(br'$-algo $\delta)$ $br$-algo $\delta$ $br'$-\alpha
(proof)

theorem $br'$-while-algo:
finite $\delta \implies$ while-algo $(br'$-algo $\delta)$
(proof)

lemma fst-br'$-\alpha$: $\text{fst} (br'$-\alpha $s) = \text{fst} s$
(proof)

theorems $br'$-invar-final = $br'$-pref.transfer-correctness[OF $br$-invar-final, unfolded $fst-br'$-\alpha]

theorem bre'$-pref-br$: wa-precise-refine $(bre'$-algo $\delta$) $(bre$-algo $\delta$) $br'$-\alpha
(proof)

interpretation bre'$-pref$: wa-precise-refine $(bre'$-algo $\delta$) $(bre$-algo $\delta$) $br'$-\alpha
(proof)

theorem bre'$-while-algo$:
finite $\delta \implies$ while-algo $(bre'$-algo $\delta$)
(proof)

theorems bre'$-invar-final = $bre'$-pref.transfer-correctness[OF bre-invar-final, unfolded $fst-br'$-\alpha]

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 $:: (\lq Q, L\rq)$ ta-rule $\Rightarrow (\lq Q, L\rq)$ $br'$-state $\Rightarrow (\lq Q, L\rq)$ $br'$-state
where
$br'$-inner-step $== \lambda r (Q,W,rcm) . c=\text{the (rcm r) in}$
$\text{if } c \leq 1 \text{ then insert (lhs r) Q else Q,}$
$\text{if } c \leq 1 \land (\text{lhs r}) \notin Q \text{ then insert (lhs r) W else W,}$
$\text{rcm ( } r \mapsto (c-(1::nat))))$
lemma \[ \text{br\textquotesingle{}-inner-invar-imp-final:} \]
\[ \begin{array}{c}
q \in W; \text{br\textquotesingle{}-inner-invar \{r \in \delta. q \in \text{set (rhsq r)}\}} \ q \ (Q,W-\{q\},\text{rcm}) \ \{} \Sigma' \}
\end{array} \]
\[ \Rightarrow (Q,W,\text{rcm},\Sigma) \in \text{br\textquotesingle{}-inner-step } \delta \]
\[ \text{(proof)} \]

lemma \[ \text{br\textquotesingle{}-inner-invar-step:} \]
\[ \begin{array}{c}
q \in W; \text{br\textquotesingle{}-inner-invar \{r \in \delta. q \in \text{set (rhsq r)}\}} \ q \ (Q,W-\{q\},\text{rcm}) \ \text{it } \Sigma';
\end{array} \]
\[ \ r \in \text{it}; \text{if } \subseteq \{r \in \delta. q \in \text{set (rhsq r)}\} \]
\[ \Rightarrow \text{br\textquotesingle{}-inner-invar \{r \in \delta. q \in \text{set (rhsq r)}\}} \ q \ (Q,W-\{q\},\text{rcm}) \]
\[ \text{(it- } r \text{)} \text{ (br\textquotesingle{}-inner-step } r \text{ } \Sigma') \]
\[ \text{(proof)} \]

lemma \[ \text{br\textquotesingle{}-inner-invar-initial:} \]
\[ \begin{array}{c}
q \in W \]
\[ \Rightarrow \text{br\textquotesingle{}-inner-invar \{r \in \delta. q \in \text{set (rhsq r)}\}} \ q \ (Q,W-\{q\},\text{rcm}) \]
\[ \{r \in \delta. q \in \text{set (rhsq r)}\} \ (Q,W-\{q\},\text{rcm}) \]
\[ \text{(proof)} \]

lemma \[ \text{br\textquotesingle{}-inner-step-proof:} \]
\[ \text{fixes } \alpha :: \Sigma \Rightarrow (Q,L) \text{ br\textquotesingle{}-state} \]
\[ \text{fixes } \text{cstep} :: (Q,L) \text{ ta-rule } \Rightarrow \Sigma \Rightarrow \Sigma \]
\[ \text{fixes } \Sigma h :: \Sigma \]
\[ \text{fixes cinvar} :: (Q,L) \text{ ta-rule set } \Rightarrow \Sigma \Rightarrow \text{bool} \]

assumes \[ \text{iterable-set; set-iteratei } \alpha \text{ invar iteratei} \]
assumes \[ \text{invar-initial: } \text{cinvar } \{r \in \delta. q \in \text{set (rhsq r)}\} \ \Sigma h \]
assumes \[ \text{invar-step:} \]
\[ \begin{array}{c}
!! it \ r \Sigma. \ {r \in \text{it}; \text{it } \subseteq \{r \in \delta. q \in \text{set (rhsq r)}\}; \text{cinvar it } \Sigma } \]
\[ \Rightarrow \text{cinvar } (\text{it- } \{r\}) \ (\text{cstep } r \ \Sigma) \]
assumes \[ \text{step-desc:} \]
\[ \begin{array}{c}
!! it \ r \Sigma. \ {r \in \text{it}; \text{it } \subseteq \{r \in \delta. q \in \text{set (rhsq r)}\}; \text{cinvar it } \Sigma } \]
\[ \Rightarrow \text{as } (\text{cstep } r \ \Sigma) = \text{br\textquotesingle{}-inner-step } r \ (\text{as } \Sigma) \]
assumes \[ \text{it-set-desc: } \text{invar it-set } \alpha \text{ it-set } \{r \in \delta. q \in \text{set (rhsq r)}\} \]

assumes \[ \text{Q IW[simp]: } q \in W \]
assumes $\Sigma$-desc[simp]: $\alpha \Sigma = (Q, W, rcm)$
assumes $\Sigma h$-desc[simp]: $\alpha \Sigma h = (Q, W - \{q\}, rcm)$

shows $(\alpha \Sigma, \alpha (iteratei it-set (\lambda\cdot True) estep \Sigma h)) \in br'\text{-step } \delta$

Computing Witnesses  

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

— A map from states to trees has the witness-property, if it maps states to trees that are accepted with that state:
definition witness-prop $\delta m == \forall q t. \ m q = \text{Some } t \rightarrow \text{accs } \delta t q$

— Construct a witness for the LHS of a rule, provided that the map contains witnesses for all states on the RHS:
definition construct-witness :: $\langle Q \Rightarrow L \text{ tree} \rangle \Rightarrow (Q, L) \text{ ta-rule } \Rightarrow L \text{ tree}$

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

lemma witness-propD: $[\text{witness-prop } \delta m; \ m q = \text{Some } t] \Rightarrow \text{accs } \delta t q$

lemma construct-witness-correct:

$[\text{witness-prop } \delta Q; \ r \in \delta; \ \text{set (rhsq r)} \subseteq \text{dom } Q ]$

$\Rightarrow \text{accs } \delta (\text{construct-witness } Q r) (\text{lhs } r)$

lemma construct-witness-eq:

$[Q \mid ^\prime \text{ set (rhsq r)} = Q' \mid ^\prime \text{ set (rhsq r)}]\Rightarrow
\text{construct-witness } Q r = \text{construct-witness } Q' r$

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

where $\text{brw-}\alpha = (\lambda (Q, W, rcm). \ (\text{dom } Q, W, rcm))$
definition brw-invar-add :: $(Q, L) \text{ ta-rule set } \Rightarrow (Q, L) \text{ brw-state set}$

where $\text{brw-invar-add } \delta == \{(Q, W, rcm), \text{ witness-prop } \delta Q\}$
definition brw-invar $\delta == \text{brw-invar-add } \delta \cap \{s. \ \text{brw-}\alpha s \in \text{br'}-\text{invar } \delta\}$

35
inductive-set \( \text{brw-step} \) :: \((\'Q,\'L)\) ta-rule set \(\Rightarrow\) \((\'Q,\'L)\) brw-state \((\'Q,\'L)\) brw-state set for \(\delta\) where

\[
\begin{align*}
q \in W; \\
d\text{sqr} = \{ r \in \delta. \ q \in \text{set} (\text{rhsq } r) \land \text{the } (\text{rcm } r) \leq 1 \}; \\
\text{dom } Q' = \text{dom } Q \cup \text{lhs}\text{sqr}; \\
!!q. \ Q' q = \text{Some } t \implies Q q = \text{Some } t \\
\land \ (\exists r \in \text{dsqr}. \ q = \text{lhs } r \land t=\text{construct-witness } Q r); \\
W' = (W-\{q\}) \cup (\text{lhs}\text{sqr} - \text{dom } Q); \\
!!r. \ r \in \delta \implies \text{rcm} r = ( \text{if } q \in \text{set} (\text{rhsq } r) \text{ then } \\
\text{Some } (\text{the } (\text{rcm } r) - 1) \\
\text{else } \text{rcm } r 
\}
\] \\
\implies ((Q,W,\text{rcm}),(Q',W',\text{rcm}') \in \text{brw-step } \delta)
\]

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

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

\[
\begin{align*}
\forall q. \ Q q = \text{Some } t \implies (\exists r \in \delta. \ \text{rhsq } r = [] \land q = \text{lhs } r \\
\land t = \text{NODE } (\text{rhsl } r) []); \\
\forall r \in \delta. \ \text{rhsq } r = [] \implies Q (\text{lhs } r) \neq \text{None} \\
\implies Q \in \text{brw-iq } \delta 
\]

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

\[
\begin{align*}
!!r. \ r \in \delta \implies \text{rcm } r = \text{Some } (\text{card } (\text{set} (\text{rhsq } r))); \ Q \in \text{brw-iq } \delta \\
\implies (Q, \text{brw-iq } \delta, \text{rcm}) \in \text{brw-initial } \delta 
\]

definition \(\text{brw-algo } Q_i \delta = [\]
\qquad \text{wa-cond} = \text{brw-cond } Q_i, \\
\qquad \text{wa-step} = \text{brw-step } \delta, \\
\qquad \text{wa-initial} = \text{brw-initial } \delta, \\
\qquad \text{wa-invar} = \text{brw-invar } \delta
\]

lemma \(\text{brw-cond-abs} : \Sigma \in \text{brw-cond } Q_i \iff (\text{brw-} \alpha \Sigma) \in \text{br}'-\text{cond } Q_i\) (proof)

lemma \(\text{brw-initial-abs} : \Sigma \in \text{brw-initial } \delta \implies \text{brw-} \alpha \Sigma \in \text{br}'-\text{initial } \delta\) (proof)

lemma \(\text{brw-invar-initial} : \text{brw-initial } \delta \subseteq \text{brw-invar-add } \delta\) (proof)
lemma \texttt{brw-step-abs}:  
\[
\{ (\Sigma,\Sigma') \in \text{brw-step} \delta \} \implies \text{brw-\alpha} \Sigma, \text{brw-\alpha} \Sigma') \in \text{br-\alpha\text{-step}} \delta
\]

(\text{proof})

lemma \texttt{brw-step-invar}:  
\begin{align*}
\text{assumes FIN[simp]: finite } \delta \\
\text{assumes INV: } \Sigma \in \text{brw-invar-add } \delta \text{ and } \text{BR'INV: } \text{brw-\alpha} \Sigma \in \text{br'-invar } \delta \\
\text{shows } \Sigma' \in \text{brw-invar-add } \delta
\end{align*}

(\text{proof})

theorem \texttt{brw-pref-bre'}: \text{wa-precise-refine} \text{brw-algo} \text{Qi } \delta \text{ (bre'-algo Qi } \delta \text{) brw-\alpha}

(\text{proof})

interpretation \texttt{brw-pref}:  
\text{wa-precise-refine} \text{brw-algo} \text{Qi } \delta \text{ (bre'-algo Qi } \delta \text{) brw-\alpha}

(\text{proof})

theorem \texttt{brw-while-algo}: \text{finite } \delta \implies \text{while-algo} \text{ (brw-algo Qi } \delta \text{)}

(\text{proof})

lemma \texttt{fst-brw-\alpha}: \text{fst} \text{ (brw-\alpha s)} = \text{dom} \text{ (fst s)}

(\text{proof})

theorem \texttt{brw-invar-final}:  
\begin{align*}
\forall \text{sc. sc } \in \text{wa-invar} \text{ (brw-algo Qi } \delta \text{)} \land \text{sc } \notin \text{wa-cond} \text{ (brw-algo Qi } \delta \text{)} \\
\implies \text{(Qi } \cap \text{ dom (fst sc)} = \text{\{\}}} \implies \text{(Qi } \cap \text{ b-accessible } \delta = \text{\{\}})
\end{align*}

\wedge \text{ (witness-prop } \delta \text{ (fst sc))}

(\text{proof})

**Implementing a Step**  
\texttt{inductive-set brw-inner-step}

\begin{align*}
\text{::} & \text{ (}'Q,'L) \text{ ta-rule } \Rightarrow \text{ (}'Q,'L) \text{ brw-state } \times \text{ (}'Q,'L) \text{ brw-state } \text{ set} \\
\text{for } r \text{ where} & \\
\text{} & \begin{cases}
\text{c } = \text{the (rcm r)}; \text{ } \Sigma = (Q,W,rcm); \text{ } \Sigma' = (Q',W',rcm') & \\
\text{if } c \leq 1 \land (\text{lhs r}) \notin \text{ dom Q then} & \\
\text{Q'} = Q(\text{lhs r } \mapsto \text{construct-witness Q r}) & \\
\text{else } Q' = Q; & \\
\text{if } c \leq 1 \land (\text{lhs r}) \notin \text{ dom Q then} & \\
\text{W'} = \text{insert (lhs r) W} & \\
\text{else W'} = W; & \\
\text{rcm'} = \text{rcm ( r } \mapsto \text{ (c } \mapsto \text{ (1::nat)))) & \\
\end{cases}
\end{align*}

\implies \text{ (}\Sigma,\Sigma') \in \text{brw-inner-step } r

\texttt{definition brw-inner-invar}

\begin{align*}
\text{::} & \text{ (}'Q,'L) \text{ ta-rule set } \Rightarrow \text{ (}'Q') \text{ brw-state } \Rightarrow \text{ (}'Q,'L) \text{ ta-rule set} \\
& \Rightarrow \text{ (}'Q,'L) \text{ brw-state } \Rightarrow \text{bool} \\
\text{where} & \\
\text{brw-inner-invar rules q } \equiv \lambda(Q,W,rcm) \text{ it } (Q',W',rcm').
\end{align*}
(\textit{br}'-inner-invar rules q (brw-\alpha (Q,W,rcm)) \iff (brw-\alpha (Q',W',rcm')) \land
\textit{br}'-dom Q = Q) \land
(let dsqr = \{ r \in rules \mid \text{the (rcm r) \leq 1} \} in
\forall q t. Q' q = \text{Some t} \rightarrow (Q q = \text{Some t} \\
\lor (Q q = \text{None} \land (\exists r \in dsqr. q = \text{lhs r} \land t = \text{construct-witness Q r}))
))

\textbf{lemma} brw-inner-step-abs:
(\Sigma,\Sigma') \in \textit{brw-inner-step} r \implies \textit{br}'-inner-step r (brw-\alpha \Sigma) = \textit{brw-\alpha} \Sigma'
(proof)

\textbf{lemma} brw-inner-invar-imp-final:
\[ q \in W; \text{brw-inner-invar} \{ r \in \delta. q \in \text{set (rqs r)} \} q (Q,W-\{q\},rcm) \{ \} \Sigma' \]
\implies ((Q,W,rcm),\Sigma') \in \textit{brw-step} \delta
(proof)

\textbf{lemma} brw-inner-invar-step:
\textbf{assumes} INVI: (Q,W,rcm) \in \textit{brw-invar} \delta
\textbf{assumes} A: q \in W \land r \in it \land \{ r \in \delta. q \in \text{set (rqs r)} \}
\textbf{assumes} INVI': \textit{brw-inner-invar} \{ r \in \delta. q \in \text{set (rqs r)} \} q (Q,W-\{q\},rcm) \land
\Sigma h
\textbf{assumes} STEP: (\Sigma h,\Sigma') \in \textit{brw-inner-step} r
\textbf{shows} \textit{brw-invar} \{ r \in \delta. q \in \text{set (rqs r)} \} q (Q,W-\{q\},rcm) (it-\{r\}) \Sigma'
(proof)

\textbf{lemma} brw-inner-invar-initial:
\[ q \in W \land \Rightarrow \textit{brw-inner-invar} \{ r \in \delta. q \in \text{set (rqs r)} \} q (Q,W-\{q\},rcm) \\
\{ r \in \delta. q \in \text{set (rqs r)} \} (Q,W-\{q\},rcm)
\]
(proof)

\textbf{theorem} brw-inner-step-proof:
\textbf{fixes} \alpha :: \langle Q',L \rangle \textit{brw-state}
\textbf{fixes} cstep :: \langle 'Q',L \rangle \textit{ta-rule} \Rightarrow \Sigma \Rightarrow \Sigma
\textbf{fixes} \Sigma h :: \Sigma
\textbf{fixes} cinvar :: \langle 'Q',L \rangle \textit{ta-rule set} \Rightarrow \Sigma \Rightarrow \textit{bool}
\textbf{assumes} set-iterate: set-iteratei \alpha invar iteratei
\textbf{assumes} invar-start: (\alpha \Sigma) \in \textit{brw-invar} \delta
\textbf{assumes} invar-initial: cinvar \{ r \in \delta. q \in \text{set (rqs r)} \} \Sigma h
\textbf{assumes} invar-step:
\[ \forall it r. \Sigma. [ r \in \textit{it}; \text{it} \subseteq \{ r \in \delta. q \in \text{set (rqs r)} \}; \text{cinvar it \Sigma} ] \]
\implies \text{cinvar (it-\{r\}) (cstep r \Sigma)}
\textbf{assumes} step-desc:
\[ \forall it r. \Sigma. [ r \in \textit{it}; \text{it} \subseteq \{ r \in \delta. q \in \text{set (rqs r)} \}; \text{cinvar it \Sigma} ] \]

38
\[
\rightarrow (\alpha r \Sigma, \alpha (cstep r \Sigma)) \in \text{brw-inner-step } r
\]

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

assumes \( QiW[simp]: q \in W \)

assumes \( \Sigma \text{-desc[simp]: } \alpha \Sigma = (Q, W, rcm) \)

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

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

\( \langle \text{proof} \rangle \)

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 Exploration in the Isabelle Collections Framework) is refined to compute the product automaton.

type-synonym \( ('Q1, 'Q2, 'L) \text{ frp-state} = ('Q1 \times 'Q2) \text{ set } \times ('Q1 \times 'Q2) \text{ list } \times (('Q1 \times 'Q2), 'L) \text{ ta-rule set} \)

definition \( \text{frp-\alpha} :: ('Q1, 'Q2, 'L) \text{ frp-state } 
\rightarrow ('Q1 \times 'Q2) \text{ dfs-state} \)

where \( \text{frp-\alpha } S = \langle (Q, W, \delta) = S \text{ in } (Q, W) \rangle \)

definition \( \text{frp-invar-add } \delta 1 \delta 2 == \)

\{ (Q, W, \delta d). \delta d = \{ r. r \in \delta - \prod \delta 1 \delta 2 \land \text{lhs } r \in Q - \text{set } W\} \}

definition \( \text{frp-invar} :: ('Q1, 'L) \text{ tree-automaton-rec } 
\rightarrow ('Q2, 'L) \text{ tree-automaton-rec} \)

\rightarrow ('Q1, 'Q2, 'L) \text{ frp-state set} \)

where \( \text{frp-invar } T1 T2 == \)

\( \text{frp-invar-add } (\text{ta-rules } T1) (\text{ta-rules } T2) \)

\cap \{ s. \text{frp-\alpha } s \in \text{dfs-invar } (\text{ta-initial } T1 \times \text{ta-initial } T2)

(f-succ (\delta - \prod (\text{ta-rules } T1) (\text{ta-rules } T2))) \}

inductive-set \( \text{frp-step} :: ('Q1, 'L) \text{ ta-rule set } 
\rightarrow ('Q2, 'L) \text{ ta-rule set} \)

\rightarrow (('Q1, 'Q2, 'L) \text{ frp-state } 
\times ('Q1, 'Q2, 'L) \text{ frp-state set}) \text{ set} \)

for \( \delta 1 \delta 2 \) where

\[ \begin{align*}
W &= (q1, q2) \# Wtl; \\
\text{distinct } Wn; \\
\text{set } Wn &= f\text{-succ } (\delta - \prod \delta 1 \delta 2)' \{ (q1, q2) \} - Q; \\
W' &= Wn@ Wtl; \\
Q' &= Q \cup f\text{-succ } (\delta - \prod \delta 1 \delta 2)' \{ (q1, q2) \}; \\
\delta d' &= \delta d \cup \{ r \in \delta - \prod \delta 1 \delta 2. \text{lhs } r = (q1, q2) \} \\
\end{align*} \]

\( \Rightarrow ((Q, W, \delta d), (Q', W', \delta d')) \in \text{frp-step } \delta 1 \delta 2 \)

inductive-set \( \text{frp-initial} :: 'Q1 \text{ set } 
\rightarrow 'Q2 \text{ set } 
\rightarrow ('Q1, 'Q2, 'L) \text{ frp-state set} \)

39
for $Q_{10} Q_{20}$ where
\[ \text{distinct } W; \text{ set } W = Q_{10} \times Q_{20} \implies (Q_{10} \times Q_{20}, W, \{\}) \in \text{frp-initial } Q_{10} Q_{20} \]

**definition** frp-cond :: (\('Q_1, 'Q_2, 'L\) frp-state set where
\[ \text{frp-cond} \equiv \{(Q, W, \delta d). W \not= \[]\} \]

**definition** frp-algo $T_1 T_2$ == ()
wa-cond = frp-cond,
wa-step = frp-step (ta-rules $T_1$) (ta-rules $T_2$),
wa-initial = frp-initial (ta-initial $T_1$) (ta-initial $T_2$),
wa-invar = frp-invar $T_1 T_2$
\]

— The algorithm refines the DFS-algorithm

**theorem** frp-pref-dfs:
wa-precise-refine (frp-algo $T_1 T_2$)
\[(dfs-algo (ta-initial $T_1$ \times ta-initial $T_2$)
(f-succ (\delta-prod (ta-rules $T_1$) (ta-rules $T_2$))))\]

**interpretation** frp-ref: wa-precise-refine (frp-algo $T_1 T_2$)
\[(dfs-algo (ta-initial $T_1$ \times ta-initial $T_2$)
(f-succ (\delta-prod (ta-rules $T_1$) (ta-rules $T_2$))))\]

**theorem** frp-white-algo:
\[\text{assumes } TA: \text{tree-automaton } T_1 \\text{tree-automaton } T_2\]
\[\text{shows } \text{while-algo } (frp-algo T_1 T_2)\]

**theorem** frp-inv-final:
\[\forall s. s \in wa-invar (frp-algo T_1 T_2) \land s \notin wa-cond (frp-algo T_1 T_2)
\implies (case s of (Q, W, \delta d) \Rightarrow
\{ ta-initial = ta-initial T_1 \times ta-initial T_2,
 ta-rules = \delta d
\} = ta-fwd-reduce (ta-prod T_1 T_2)\]

end
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

— Make rules hashable

**instantiation** ta-rule :: (hashable,hashable) hashable

**begin**

**fun** hashcode-of-ta-rule

:: ('Q1::hashable,'Q2::hashable) ta-rule ⇒ hashcode

**where**

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

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

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

**instance**

**(proof)**

**end**

— Make wrapped states hashable

**instantiation** ustate-wrapper :: (hashable,hashable) hashable

**begin**

**definition** hashcode x == (case x of USW1 a ⇒ 2 * hashcode a | USW2 b ⇒ b)

**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** ll-lll-iflt-cp: inj-image-filter-cp-loc ls-ops ls-ops ls-ops **(proof)**
interpretation hhh-cart: cart-loc hs-ops hs-ops hs-ops ⟨proof⟩
interpretation hh-set-xy!: g-set-xy-loc hs-ops hs-ops ⟨proof⟩
interpretation lh-set-xyy!: g-set-xyy-loc ls-ops ls-ops hs-ops ⟨proof⟩
interpretation hh-map-to-nat!: map-to-nat-loc hs-ops hm-ops ⟨proof⟩
interpretation hh-set-xy!: g-set-xy-loc hs-ops hs-ops ⟨proof⟩
interpretation lh-set-xy!: g-set-xy-loc ls-ops hs-ops ⟨proof⟩
interpretation hh-set-xx!: g-set-xx-loc hs-ops hs-ops ⟨proof⟩
interpretation hs-to-fifo!: set-to-list-loc hs-ops fifo-ops ⟨proof⟩

⟨ML⟩

5.2 Generating Indices of Rules

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

f index of rules by function symbol

s index of rules by lhs

sf index of rules

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

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

lemma build-rule-index-f-correct[simp]:
assumes I[simp, intro!]: ls-invar δ
shows hll-idx.is-index rhs (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
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.

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

  — Abstraction of a concrete tree automaton to an abstract one

definition hta-α
  where hta-α H = |
    ta-initial = hs-α (hta-Qi H), ta-rules = ls-α (hta-δ H)

  — Builds the f-index if not present

definition hta-ensure-idx-f H ==
  case hta-idx-f H of
    None ⇒ H |
    Some - ⇒ H

  — Builds the s-index if not present

definition hta-ensure-idx-s H ==
  case hta-idx-s H of
    None ⇒ H |
    Some - ⇒ H

  — Builds the sf-index if not present

definition hta-ensure-idx-sf H ==
  case hta-idx-sf H of
    None ⇒ H |
    Some - ⇒ H

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

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

43
\begin{proof}

**Lemma** \( \text{hta-ensure-idx-sf-correct\text{-}\(\alpha\)[simp]} \):

\[ \text{hta-\(\alpha\)} (\text{hta-ensure-idx-sf \(H\)}) = \text{hta-\(\alpha\)} \(H\) \]

\end{proof}

\begin{proof}

**Lemma** \( \text{hta-ensure-idx-other}[simp] \):

\[ \text{hta-\(Q_i\)} (\text{hta-ensure-idx-f \(H\)}) = \text{hta-\(Q_i\)} \(H\) \]

\[ \text{hta-\(\delta\)} (\text{hta-ensure-idx-f \(H\)}) = \text{hta-\(\delta\)} \(H\) \]

\[ \text{hta-\(Q_i\)} (\text{hta-ensure-idx-s \(H\)}) = \text{hta-\(Q_i\)} \(H\) \]

\[ \text{hta-\(\delta\)} (\text{hta-ensure-idx-s \(H\)}) = \text{hta-\(\delta\)} \(H\) \]

\end{proof}

**Definition** \( \text{hta-has-idx-f \(H\)} \equiv \text{hta-idx-f \(H\)} \neq \text{None} \)

— Check whether the s-index is present

**Definition** \( \text{hta-has-idx-s \(H\)} \equiv \text{hta-idx-s \(H\)} \neq \text{None} \)

— Check whether the sf-index is present

**Definition** \( \text{hta-has-idx-sf \(H\)} \equiv \text{hta-idx-sf \(H\)} \neq \text{None} \)

\begin{proof}

**Lemma** \( \text{hta-idx-f-pres} \)

[simp, intro!]: \( \text{hta-has-idx-f \(H\)} \equiv \text{hta-idx-f \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-s \(H\)} \equiv \text{hta-idx-s \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-sf \(H\)} \equiv \text{hta-idx-sf \(H\)} \neq \text{None} \)

\end{proof}

\begin{proof}

**Lemma** \( \text{hta-idx-s-pres} \)

[simp, intro!]: \( \text{hta-has-idx-s \(H\)} \equiv \text{hta-idx-s \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-f \(H\)} \equiv \text{hta-idx-f \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-sf \(H\)} \equiv \text{hta-idx-sf \(H\)} \neq \text{None} \)

\end{proof}

\begin{proof}

**Lemma** \( \text{hta-idx-sf-pres} \)

[simp, intro!]: \( \text{hta-has-idx-sf \(H\)} \equiv \text{hta-idx-sf \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-f \(H\)} \equiv \text{hta-idx-f \(H\)} \neq \text{None} \) and

[simp, intro]: \( \text{hta-has-idx-s \(H\)} \equiv \text{hta-idx-s \(H\)} \neq \text{None} \)

\end{proof}

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

— Lookup rules by function symbol

**Definition** \( \text{hta-lookup-f \(f\) \(H\)} \equiv \text{tll-idx.lookup \(f\) \(\text{the (hta-idx-f \(H\)})\)} \)

— Lookup rules by lhs-state

**Definition** \( \text{hta-lookup-s \(q\) \(H\)} \equiv \text{tll-idx.lookup \(q\) \(\text{the (hta-idx-s \(H\)})\)} \)

— Lookup rules by function symbol and lhs-state

**Definition** \( \text{hta-lookup-sf \(q\) \(f\) \(H\)} \equiv \text{tll-idx.lookup \((q,f)\) \(\text{the (hta-idx-sf \(H\)})\)} \)

44
locale hashedTa =
  fixes H :: ('Q::hashable,'L::hashable) hashedTa

locale}
automaton

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

end

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

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

— Building a tree automaton from a valid tree automaton yields again a valid
  tree automaton. This operation has the only effect of removing the indices.
lemma (in hashedTa) init-hta-is-hta:
hashedTa (init-hta (hta-Qi H) (hta-δ H))
⟨proof⟩

5.4 Algorithm for the Word Problem

lemma r-match-by-laz: r-match L l = list-all-zip (λQ q ∈ Q) L l
⟨proof⟩

Executable function that computes the set of accepting states for a given
tree
fun faccs′ where
faccs′ H (NODE f ts) = (let Qs = List.map (faccs′ H) ts in
  ll-set-xy.g-image-filter (λr. case r of (q → f′ qs) ⇒
    if list-all-zip (λQ q. ls-memb q Q) Qs qs then Some (lhs r) else None
  )
  (hta-lookup-f f H)
)

— Executable algorithm to decide the word-problem. The first version depends
  on the f-index to be present, the second version computes the index if not
  present.
definition hta-mem′ t H == ¬ll-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)
  \text{list-all ls-invar } (\text{List.map (faccs′ H) ts}) (is ?T2)
⟨\text{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)
  \text{List.map ls-\alpha } (\text{List.map (faccs′ H) ts})
  = \text{List.map (faccs (ls-\alpha (hta-\delta H))) ts (is ?T2)}
⟨\text{proof}⟩

lemma hta-mem′-correct:
hta-has-idx-f H \Rightarrow hta-mem′ t H \longleftrightarrow t \in \text{ta-lang } (hta-\alpha H)
⟨\text{proof}⟩

theorem hta-mem-correct: hta-mem t H \longleftrightarrow t \in \text{ta-lang } (hta-\alpha H)
⟨\text{proof}⟩

end

5.5 Product Automaton and Intersection

5.5.1 Brute Force Product Automaton

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

definition δ-prod-h
  :: ('q1::hashable,'l::hashable) ta-rule ls
  ⇒ ('q2::hashable,'l) ta-rule ls ⇒ ('q1×'q2,'l) ta-rule ls

where δ-prod-h δ1 δ2 ==
lit-iff-cp.inj-image-filter-cp (λ(r1,r2). r-prod r1 r2)
  (λ(r1,r2). rhs l1 = rhs l2
  ∧ length (rhsq r1) = length (rhsq r2))
δ1 δ2

lemma r-prod-inj:
[ rhs l1 = rhs l2; length (rhsq r1) = length (rhsq r2); rhs l1′ = rhs l2′; length (rhsq r1′) = length (rhsq r2′); r-prod r1 r2 = r-prod r1′ r2′ ] \Rightarrow r1=r1′ ∧ r2=r2′
\begin{proof}

\begin{lemma} \( \delta\text{-prod-h-correct} \):
\begin{assumes}
\( INV[J\text{simp;} \text{ls-invar } \delta 1 \) \text{ ls-invar } \delta 2 \)
\end{assumes}
\begin{shows}
\( \text{ls-}\alpha (\delta\text{-prod-h } \delta 1 \delta 2) = \delta\text{-prod} (\text{ls-}\alpha \delta 1) (\text{ls-}\alpha \delta 2) \)
\( \text{ls-invar} (\delta\text{-prod-h } \delta 1 \delta 2) \)
\end{shows}
\end{lemma}
\begin{proof}

\begin{definition}
\( \text{hta-prodWR } H 1 H 2 \) ==
\( \text{init-hta} \ (\text{hhh-cart.cart} (\text{hta-Qi } H 1) (\text{hta-Qi } H 2)) \ (\delta\text{-prod-h} \ (\text{hta-}\delta H 1) \ (\text{hta-}\delta H 2)) \)
\end{definition}

\begin{lemma} \( \text{hta-prodWR-correct-aux} \):
\begin{assumes}
\( A: \text{hashedTa } H 1 \) \text{ hashedTa } H 2 \)
\end{assumes}
\begin{shows}
\( \text{hta-}\alpha (\text{hta-prodWR } H 1 H 2) = \text{ta-prod} (\text{hta-}\alpha H 1) (\text{hta-}\alpha H 2) (\text{ls } ?T 1) \)
\( \text{hashedTa} (\text{hta-prodWR } H 1 H 2) (\text{ls } ?T 2) \)
\end{shows}
\end{lemma}
\begin{proof}

\begin{lemma} \( \text{hta-prodWR-correct} \):
\begin{assumes}
\( TA: \text{hashedTa } H 1 \) \text{ hashedTa } H 2 \)
\end{assumes}
\begin{shows}
\( \text{ta-lang} (\text{hta-}\alpha (\text{hta-prodWR } H 1 H 2)) \)
\( \text{ta-lang} (\text{hta-}\alpha H 1) \cap \text{ta-lang} (\text{hta-}\alpha H 2) \)
\( \text{hashedTa} (\text{hta-prodWR } H 1 H 2) \)
\end{shows}
\end{lemma}
\begin{proof}

\end{proof}

\end{proof}

\end{proof}

\section{Product Automaton with Forward-Reduction}

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

— State of the product-automaton DFS-algorithm
\begin{type-synonym}
\( ('q 1 , 'q 2 , 'l) \text{ pa-state} \\
= ('q 1 \times 'q 2) \text{ hs} \times ('q 1 \times 'q 2) \text{ list} \times ('q 1 \times 'q 2, 'l) \text{ ta-rule ls} \)
\end{type-synonym}

— Abstraction mapping to algorithm specified in Section 4.
\begin{definition}
\( \text{pa-}\alpha \)
\( :: ('q 1 :: \text{hashable}, 'q 2 :: \text{hashable}, 'l :: \text{hashable}) \text{ pa-state} \\
\Rightarrow ('q 1 , 'q 2 , 'l) \text{ frp-state} \)
\begin{where}
\( \text{pa-}\alpha S \) \( \Rightarrow \) \( \text{let} \ (Q, W, \delta d)=S \text{ in} \ (\text{hs-}\alpha Q, W, \text{ls-}\alpha \delta d) \)
\end{where}
\end{definition}

\begin{definition}
\( \text{pa-cond} \)
\( :: ('q 1 :: \text{hashable}, 'q 2 :: \text{hashable}, 'l :: \text{hashable}) \text{ pa-state} \Rightarrow \text{bool} \)
\begin{where}
\( \text{pa-cond} S \) \( \Rightarrow \) \( \text{let} \ (Q, W, \delta d) = S \text{ in} \ W \not= [] \)
\end{where}
\end{definition}

— Adds all successor states to the set of discovered states and to the worklist
\begin{fun}
\( \text{pa-apd-rule} \\
:: ('q 1 \times 'q 2) \text{ hs} \Rightarrow ('q 1 \times 'q 2) \text{ list} \)
\end{fun}

48
⇒ (('q1::hashable) × ('q2::hashable)) list
⇒ (('q1 × 'q2) hs × ('q1 × 'q2) list)

where
pa-upd-rule Q W [] = (Q, W) |
pa-upd-rule Q W (qp#qs) = (if hs-memb qp Q then
  pa-upd-rule (hs-ins qp Q) (qp#W) qs
  else pa-upd-rule Q W qs)
)

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

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

definition pa-invar-add::
  ('q1::hashable, 'q2::hashable, 'l::hashable) pa-state set
where pa-invar-add == { (Q, W, δd). hs-invar Q ∧ ls-invar δd }
definition \( \text{pa-invar } H1 \ H2 = \)
\( \text{pa-invar-add} \cap \{ s. (\text{pa} \cdot s) \in \text{frp-invar} (hta \cdot H1) (hta \cdot H2) \} \)

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

lemma \( \text{pa-upd-rule-correct} : \)
assumes \( \text{INV} [\text{simp, intro}] : \text{hs-invar } Q \)
assumes \( \text{FMT} : \text{pa-upd-rule } Q \ W \ qs = (Q', W') \)
shows \( \text{hs-invar } Q' (\text{is } ?T1) \)
\( \text{hs-} \alpha \ Q' = \text{hs-} \alpha \ Q \cup \text{set } qs (\text{is } ?T2) \)
\( \exists Wn. \text{distinct } Wn \ \& \ \text{set } Wn = \text{set } qs - \text{hs-} \alpha \ Q \ \& \ W' = Wn@W \ (\text{is } ?T3) \)
(\text{proof})

lemma \( \text{pa-step-correct} : \)
assumes \( \text{TA} : \text{hashedTa } H1 \ \text{hashedTa } H2 \)
assumes \( \text{idx} [\text{simp}] : \text{hta-has-idx-s } H1 \ \text{hta-has-idx-sf } H2 \)
assumes \( \text{INV} : (Q, W, \delta, d) \in \text{pa-invar } H1 \ H2 \)
assumes \( \text{COND} : \text{pa-cond } (Q, W, \delta, d) \)
shows \( (\text{pa-step } H1 \ H2 (Q, W, \delta, d)) \in \text{pa-invar-add} (\text{is } ?T1) \)
\( (\text{pa-} \alpha (Q, W, \delta, d), \text{pa-} \alpha (\text{pa-step } H1 \ H2 (Q, W, \delta, d))) \)
\( \in \text{frp-step } (ls- \alpha (hta-\delta H1)) (ls- \alpha (hta-\delta H2)) (\text{is } ?T2) \)
(\text{proof})

lemma \( \text{pa-pref-frp} : \)
assumes \( \text{TA} : \text{hashedTa } H1 \ \text{hashedTa } H2 \)
assumes \( \text{idx} [\text{simp}] : \text{hta-has-idx-s } H1 \ \text{hta-has-idx-sf } H2 \)
shows \( \text{wa-precise-refine } (\text{det-wa-wa } (\text{pa-det-algo } H1 \ H2)) \)
\( (\text{frp-algo } (hta-\alpha H1) (hta-\alpha H2)) \)
\( \text{pa-} \alpha \)
(\text{proof})

lemma \( \text{pa-while-algo} : \)
assumes \( \text{TA} : \text{hashedTa } H1 \ \text{hashedTa } H2 \)
assumes \( \text{idx} [\text{simp}] : \text{hta-has-idx-s } H1 \ \text{hta-has-idx-sf } H2 \)
shows \( \text{while-algo } (\text{det-wa-wa } (\text{pa-det-algo } H1 \ H2)) \)
(\text{proof})

lemmas \( \text{pa-det-while-algo} = \text{det-while-algo-intro} [\text{OF pa-while-algo}] \)
— Transferred correctness lemma

lemmas \( \text{pa-inv-final} = \)
\( \text{wa-precise-refine.transfer-correctness} [\text{OF pa-pref-frp frp-inv-final}] \)

50
— 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,δ_d) = while pa-cond (pa-step H1 H2) (pa-initial H1 H2) in
    init-hta (hhh-cart.cart (hta-Qi H1) (hta-Qi H2)) δ_d

definition hta-prod H1 H2 ==
    hta-prod’ (hta-ensure-idx-s H1) (hta-ensure-idx-sf H2)

lemma hta-prod’-correct-aux:
    assumes TA: hshdTa H1 hshdTa H2
    assumes idx: hta-has-idx-s H1 hta-has-idx-sf H2
    shows hta-α (hta-prod’ H1 H2)
        = ta-fwd-reduce (ta-prod (hta-α H1) (hta-α H2)) (is ?T1)
        hashedTa (hta-prod’ H1 H2) (is ?T2)
  ⟨proof⟩

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

lemma hta-prod-correct-aux:
    assumes TA[simp]: hshdTa H1 hshdTa H2
    shows hta-α (hta-prod H1 H2) = ta-fwd-reduce (ta-prod (hta-α H1) (hta-α H2))
        hashedTa (hta-prod H1 H2)
  ⟨proof⟩

theorem hta-prod-correct:
    assumes TA: hshdTa H1 hshdTa H2
    shows ta-lang (hta-α (hta-prod H1 H2))
        = ta-lang (hta-α H1) ∩ ta-lang (hta-α H2)
        hashedTa (hta-prod H1 H2)
  ⟨proof⟩
5.6 Remap States
— Mapping the states of an automaton

**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-δ H))

**lemma** (in hashedTa) hta-remap-correct:
\[
shows \ hta-α (hta-remap f H) = ta-remap f (hta-α H)
\]
\[
hashedTa (hta-remap f H)
\]
\[
⟨ \text{proof} \rangle
\]

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.

— Set of states of a rule

**fun** rule-states-l **where**
\[
rule-states-l (q \rightarrow f qs) = ls-ins q (ls.from-list qs)
\]

**lemma** rule-states-l-correct[simp]:
\[
ls-α (rule-states-l r) = rule-states r
\]
\[
ls-invar (rule-states-l r)
\]
\[
⟨ \text{proof} \rangle
\]

**definition** hta-δ-states H ==
(lh-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:
\[
hs-α (hta-δ-states H) = δ-states (ta-rules (hta-α H))
\]
\[
hs-invar (hta-δ-states H)
\]
\[
⟨ \text{proof} \rangle
\]

**lemma** (in hashedTa) hta-states-correct:
\[
hs-α (hta-states H) = ta-rstates (hta-α H)
\]
\[
hs-invar (hta-states H)
\]
\[
⟨ \text{proof} \rangle
\]

**definition** reindex-map H ==
\[
\lambda q. \ the \ (hm-lookup q \ (hh-map-to-nat.map-to-nat \ (hta-states H)))
\]
definition hta-reindex
:: (′Q::hashable,′L::hashable) hashedTa ⇒ (nat,′L) hashedTa where
hta-reindex H == hta-remap (reindex-map H) H
declare hta-reindex-def [code del]

— This version is more efficient, as the map is only computed once

lemma [code]: hta-reindex H = (let mp = (hh-map-to-nat.map-to-nat (hta-states H)) in
hta-remap (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)
  hta-lang (hta-reindex H)
⟨proof⟩

5.7 Union

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

lemma hta-union-correct':
assumes TA: hta-reindex H1 hta-reindex H2
shows hta-α (hta-union H1 H2)
  = ta-union-wrap (hta-α H1) (hta-α H2) (is ?T1)
  hta-lang (hta-union H1 H2) (is ?T2)
⟨proof⟩

theorem hta-union-correct:
assumes TA: hta-reindex H1 hta-reindex H2
shows
  ta-lang (hta-α (hta-union H1 H2))
  = ta-lang (hta-α H1) ∪ ta-lang (hta-α H2) (is ?T1)
  hta-lang (hta-union H1 H2) (is ?T2)
5.8 Operators to Construct Tree Automata

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

— The empty automaton
definition hta-empty :: unit ⇒ ('q::hashable,'l::hashable) hashedTa
  where hta-empty u == init-hta (hs-empty ()) (ls-empty ())
lemma hta-empty-correct [simp, intro!]:
  shows (hta-α (hta-empty ())) = ta-empty
  hashedTa (hta-empty ())
⟨proof⟩
definition hta-add-qi :: 'q ⇒ ('q::hashable,'l::hashable) hashedTa ⇒ ('q,'l) hashedTa
  where hta-add-qi qi H == init-hta (hs-ins qi (hta-Qi H)) (hta-δ H)
lemma (in hashedTa) hta-add-qi-correct [simp, intro!]:
  shows (hta-α (hta-add-qi qi H))
    = (| ta-initial = insert qi (ta-initial (hta-α H)),
     ta-rules = ta-rules (hta-α H)
    |) hashedTa (hta-add-qi qi H)
  ⟨proof⟩
lemmas [simp, intro] = hashedTa.hta-add-qi-correct

— Add a rule to the automaton
definition hta-add-rule :: ('q,'l) ta-rule ⇒ ('q::hashable,'l::hashable) hashedTa
  ⇒ ('q,'l) hashedTa
  where hta-add-rule r H == init-hta (hta-Qi H) (ls-ins r (hta-δ H))
lemma (in hashedTa) hta-add-rule-correct [simp, intro!]:
  shows (hta-α (hta-add-rule r H))
    = (| ta-initial = ta-initial (hta-α H),
     ta-rules = insert r (ta-rules (hta-α H))
    |) hashedTa (hta-add-rule r H)
  ⟨proof⟩
lemmas [simp, intro] = hashedTa.hta-add-rule-correct

— Reduces an automaton to the given set of states
definition hta-reduce H Q ==
  init-hta (hs-inter Q (hta-Qi H))
  (ll-set-xy,g-image-filter)
\[(\lambda r. \text{if } \text{hs-memb} (\text{lhs } r) Q \land \text{list-all} (\lambda q. \text{hs-memb} q Q) (\text{rhs} q r) \text{ then } \text{Some } r \text{ else None}) (hta-\delta H)\]

**theorem** (in hashedTa) hta-reduce-correct:
  assumes INV[simp]: hs-invar Q
  shows
  hta-\alpha (hta-reduce H Q) = ta-reduce (hta-\alpha H) (hs-\alpha Q) (is ?T1)
  hashedTa (hta-reduce H Q) (is ?T2)
  (proof)

### 5.9 Backwards Reduction and Emptiness Check

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

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

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

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

— Whether the index satisfies the map and set invariants
**definition** rqrm-invar rqrm ==
 hm-invar rqrm \land (\forall q. ls-invar (rqrm-lookup rqrm q))
— Whether the index really maps a state to the set of rules with this state on their rhs
**definition** rqrm-prop \delta rqrm ==
 \forall q. ls-\alpha (rqrm-lookup rqrm q) = \{ r \in \delta. q \in \text{set} (rhsq r) \}
lemma rqrm-α-lookup-update[simp]:
\[ \text{ls-α (rqrm-lookup (rqrm-add q r rqrm) q')} =\]
\begin{align*}
&= \left( \text{if } q = q' \text{ then } \\
&\quad \text{insert } r \text{ (ls-α (rqrm-lookup rqrm q')) } \\
&\text{else } \\
&\quad \text{ls-α (rqrm-lookup rqrm q') } \right)
\end{align*}
⟨proof⟩

lemma rqrm-propD:
\[ \text{rqrm-prop δ rqrm } \Rightarrow \text{ls-α (rqrm-lookup rqrm q)} = \{ r \in δ. q \in \text{set (rhsq r)} \} \]
⟨proof⟩

lemma build-rqrm-correct:
fixes δ
assumes \[ \text{ls-invar δ} \]
shows \[ \text{rqrm-invar (build-rqrm δ) (is ?T1) and} \]
\[ \text{rqrm-prop (ls-α δ) (build-rqrm δ) (is ?T2)} \]
⟨proof⟩

type-synonym \((Q,L)\) brc-state
\[ = 'Q \times 'Q \times (('Q,L) ta-rule, nat) \times \]
— Abstraction to α’-level:
definition brc-α :: \((Q::hashable,L::hashable)\) brc-state ⇒ \((Q,L)\) br'-state
where brc-α == \(λ(Q,W,rcm). (hs-α Q, set W, hm-α rcm)\)
definition brc-invar-add :: \((Q::hashable,L::hashable)\) brc-state set
where brc-invar-add == \{(Q,W,rcm). \hs-invar Q \land \}
\[ \text{distinct } W \land \]
\[ \text{hm-invar rcm} \]
\[ (\ast \land \text{set } W \subseteq \text{hs-α } Q \star)\}
definition brc-invar δ == brc-invar-add ∩ \{ s. \text{brc-α } s \in \text{br'-invar δ} \}
definition brc-cond :: \((Q::hashable,L::hashable)\) brc-state ⇒ bool
where brc-cond == \(λ(Q,W,rcm). W \neq []\)
definition brc-inner-step :: \((q,l)\) ta-rule ⇒ \((q',l')\) brc-state
⇒ \((q,l)\) brc-state
where brc-inner-step r == \(λ(Q,W,rcm). \]
\[ \text{let } c = \text{the (hm-lookup r rcm)}; \]

56
rcm’ = hm-update r (c−(1::nat)) rcm;
Q’ = (if c ≤ 1 then hs-ins (lhs r) Q else Q);
W’ = (if c ≤ 1 ∧ ¬ hs-memb (lhs r) Q then lhs r # W else W) in
(Q’, W’, rcm’)

definition brc-step
:: ('q,'q,'l) ta-rule ls hm
⇒ ('q::hashable,'l::hashable) brc-state
⇒ ('q,'l) brc-state

where
brc-step rqrm == λ(Q, W, rcm).
ls-iteratei (rqrm-lookup rqrm (hd W)) (λ-. True) brc-inner-step
(Q, tl W, rcm)

— Initial concrete state
definition brc-iq :: ('q,'l) ta-rule ls ⇒ 'q::hashable hs

where brc-iq δ == lh-set-xy.g-image-filter (λr.
if rhsq r = [] then Some (lhs r) else None) δ

definition brc-rcm-init
:: ('q::hashable,'l::hashable) ta-rule ls
⇒ (('q,'l) ta-rule,nat) hm

where brc-rcm-init δ ==
ls-iteratei δ (λ-. True)
(λr res. hm-update r ((length (remdups (rhsq r)))) res)
(hm-empty ()))

definition brc-initial
:: ('q::hashable,'l::hashable) ta-rule ls ⇒ ('q,'l) brc-state

where brc-initial δ ==
let iq = brc-iq δ in
(iq, hs-to-list (iq), brc-rcm-init δ)

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

— Additional facts needed from the abstract level
lemma brc-ineq-wrap br-invar δ ⇒ set W ⊆ hs-α Q
⟨proof⟩

lemma brc-iq-correct:
assumes [simp]: ls-invar δ
shows hs-invar (brc-iq δ)
hs-α (brc-iq δ) = br-iq (ls-α δ)
⟨proof⟩
lemma brc-rcm-init-correct:
assumes INV[simp]: ls-invar δ
shows r ∈ ls-α δ
  \implies hm-α (brc-rcm-init δ) r = Some ((\text{card} (\text{set} \text{rhsq} r)))
(is - \implies ?T1 r) and
hm-invar (brc-rcm-init δ) (is ?T2)
⟨proof⟩

lemma brc-inner-step-br′-desc:
\exists (Q,W,rcm) ∈ brc-invar δ \implies brc-α (brc-inner-step r (Q,W,rcm)) = (is if the (hm-α rcm r) \leq 1 then insert (lhs r) (hs-α Q)
else hs-α Q,
if the (hm-α rcm r) \leq 1 \land (lhs r) \notin hs-α Q then insert (lhs r) (set W)
else (set W),
((hm-α rcm)(r \mapsto \text{the} (hm-α rcm r) - 1))
) ⟨proof⟩

lemma brc-step-invar:
assumes RQRM: rqrm-invar rqrm rqrm-prop δ rqrm
shows [Σ ∈ brc-invar-add; brc-α Σ ∈ br′-invar δ; brc-cond Σ ]
\implies (brc-step rqrm Σ) ∈ brc-invar-add
⟨proof⟩

lemma brc-step-abs:
assumes RQRM: rqrm-invar rqrm rqrm-prop δ rqrm
assumes A: Σ ∈ brc-invar δ brc-cond Σ
shows (brc-α Σ, brc-α (brc-step rqrm Σ)) ∈ br′-step δ
⟨proof⟩

lemma brc-initial-invar: ls-invar δ \implies (brc-initial δ) ∈ brc-invar-add
⟨proof⟩

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

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

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

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

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

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

definition \texttt{hta-bwd-reduce} \texttt{H} ==
let \texttt{rqrm} = build-rqrm (\texttt{hma}\(\delta\) \texttt{H}) in
\texttt{hta-reduce} \texttt{H}
(fst (while \texttt{brc-cond (brc-step rqrm) (brc-initial (hma-\(\delta\) \texttt{H}))}))

theorem (in \texttt{hashedTa}) \texttt{hta-bwd-reduce-correct}:
\textbf{shows} \texttt{hma}\(\alpha\) (\texttt{hta-bwd-reduce} \texttt{H})
= \texttt{ta-reduce} (hma-\(\alpha\) \texttt{H}) (b-accessible (ls-\(\alpha\) (hma-\(\delta\) \texttt{H}))) \texttt{is} \texttt{\texttt{T1}}
hashedTa (hta-bwd-reduce \texttt{H}) \texttt{is} \texttt{\texttt{T2}}
\langle\text{proof}\rangle

5.9.1 Emptiness Check with Witness Computation

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

lemma \texttt{brec-construct-witness-correct}:
[\texttt{hma-invar} \texttt{Qm}] \Rightarrow
\texttt{brec-construct-witness} \texttt{Qm} \texttt{r} = \texttt{construct-witness} (hma-\(\alpha\) \texttt{Qm} \texttt{r})
\langle\text{proof}\rangle

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

— Abstractions

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

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

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

definition brec-igm :: ('q::hashable, 'l::hashable) ta-rule ls \Rightarrow ('q, 'l tree) hm
where brec-igm \delta ==
ls-iteratei \delta (\lambda -. True) (\lambda r m. if rhsq r = [] then
hm-update (lhs r) (NODE (rhsl r) []) m
else m)
(hm-empty ())

definition brec-initial :: 'q hs \Rightarrow ('q::hashable, 'l::hashable) ta-rule ls
\Rightarrow ('q, 'l) brec-state
where brec-initial Qi \delta ==
let iq = brec-igm \delta in
(brec-igm \delta, hs-to-fifo.g-set-to-listr iq,
definition brec-cond
:: ('q, l) brec-state ⇒ bool
where brec-cond == λ(Q, W, rcm, qwit). ~ fifo-isEmpty W ∧ qwit = None

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

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

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

lemma brec-invar-cons:
Σ ∈ brec-invar Qi δ ⇒ Σ ∈ brec-invar-inner Qi
⟨proof⟩

lemma brec-brw-invar-cons:
  brec-α Σ ∈ brw-invar Qi ⇒ set (fifo-α (snd Σ)) ⊆ dom (hm-α (fst Σ))
⟨proof⟩

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

lemma brec-iqm-correct':
  assumes INV[simp]: ls-invar δ
  shows
    dom (hm-α (brec-iqm δ)) = {lhs r | r. r∈ls-α δ ∧ rhsq r = []} (is ?T1)
    witness-prop (ls-α δ) (hm-α (brec-iqm δ)) (is ?T2)
    hm-invar (brec-iqm δ) (is ?T3)
    ⟨proof⟩

lemma brec-iqm-correct:
  assumes INV[simp]: ls-invar δ
shows $hm$-$\alpha$ ($brc$-$iq$ $\delta$) $\in$ $brw$-$iq$ ($ls$-$\alpha$ $\delta$)

(proof)

lemma brec-inner-step-brw-desc:

\[
[ \Sigma \in brec$-$inner$-$inner$ ($hs$-$\alpha$ $Qi$) ]
\implies (brec-$\alpha$ $\Sigma$, brec-$\alpha$ ($brec$-$inner$-$step$ $Qi$ $r$ $\Sigma$)) $\in$ $brw$-$inner$-$step$ $r$
\]

(proof)

lemma brec-step-invar:

assumes $RQRM$: $rqrm$-$invar$ $rqrm$ $rqrm$-$prop$ $\delta$ $rqrm$
assumes $INV[simp]$: $hs$-$invar$ $Qi$
shows $\Sigma \in brec$-$invar$-$add$ ($hs$-$\alpha$ $Qi$); brec-$\alpha$ $\Sigma$ $\in$ $brw$-$invar$ $\delta$; brec-cond $\Sigma$
\implies (brec-step $rqrm$ $Qi$ $\Sigma$)$\in$brec-$invar$-$add$ ($hs$-$\alpha$ $Qi$)

(proof)

lemma brec-invar-initial:

\[
[l$-$invar$ $\delta$; $hs$-$invar$ $Qi$] $\implies$ (brec-initial $Qi$ $\delta$) $\in$ $brec$-$invar$-$add$ ($hs$-$\alpha$ $Qi$
\]

(proof)

lemma brec-cond-abs:

\[
[\Sigma \in brec$-$invar$ $Qi$ $\delta$] $\implies$ brec-cond $\Sigma$ $\leftarrow$ (brec-$\alpha$ $\Sigma$)$\in$brw-cond $Qi$
\]

(proof)

lemma brec-initial-abs:

\[
[l$-$invar$ $\delta$; $hs$-$invar$ $Qi$ ]
\implies$ brec-$\alpha$ (brec-initial $Qi$ $\delta$) $\in$ $brw$-$initial$ ($ls$-$\alpha$ $\delta$
\]

(proof)

lemma brec-pref-brw:

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

(proof)

lemma brec-while-algo:

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

\textbf{lemma} \  \text{fst-brec-} \alpha: \text{fst} (\text{brec-} \alpha \ \Sigma) = \text{hm-} \alpha (\text{fst} \ \Sigma)

\langle \text{proof} \rangle

\textbf{lemmas} \ \text{brec-invar-final} =
\text{wa-precise-refine.transfer-correctness}
\text{OF brec-pref-brw brw-invar-final,}
\text{unfolded \text{fst-brec-} \alpha}

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

\textbf{definition} \ \text{hta-is-empty-witness} \ H ==
\text{let rqrm = build-rqrm (hta-} \delta \ \Sigma)\ ;
\text{(Q, -, -qwit)} = (\text{while brec-cond (brec-step rqrm (hta-Qi} \ H))}
\text{(brec-initial (hta-Qi} \ H) (hta-} \delta \ \Sigma))\\
in
\text{case qwit of}
\quad \text{None } \Rightarrow \text{None} \mid
\quad \text{Some } q \Rightarrow (\text{hm-lookup } q Q)

\textbf{theorem} \ (\text{in hashedTa}) \ \text{hta-is-empty-witness-correct:}
\text{shows [rule-format]: hta-is-empty-witness} \ H = \text{Some } t
\quad \rightarrow \ t \in \text{ta-lang (hta-} \alpha \ \Sigma) \ (\text{is } \ ?T1)
\text{hta-is-empty-witness} \ H = \text{None } \rightarrow \text{ta-lang (hta-} \alpha \ \Sigma) = \{} \ (\text{is } \ ?T2)

\langle \text{proof} \rangle

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.

\textbf{type-synonym} \ \text{htai} = (\text{nat,nat}) \text{ hashedTa}

\textbf{definition} \ \text{htai-mem} :: - \Rightarrow \text{htai} \Rightarrow \text{bool}
\text{where} \ \text{htai-mem} == \text{hta-mem}

\textbf{definition} \ \text{htai-prod} :: \text{htai} \Rightarrow \text{htai} \Rightarrow \text{htai}
\text{where} \ \text{htai-prod} \ H1 \ H2 == \text{hta-reindex (hta-prod} \ H1 \ H2)

\textbf{definition} \ \text{htai-prodWR} :: \text{htai} \Rightarrow \text{htai} \Rightarrow \text{htai}
\text{where} \ \text{htai-prodWR} \ H1 \ H2 == \text{hta-reindex (hta-prodWR} \ H1 \ H2)

\textbf{definition} \ \text{htai-union} :: \text{htai} \Rightarrow \text{htai} \Rightarrow \text{htai}
\text{where} \ \text{htai-union} \ H1 \ H2 == \text{hta-reindex (hta-union} \ H1 \ H2)

\textbf{definition} \ \text{htai-empty} :: \text{unit} \Rightarrow \text{htai}
\text{where} \ \text{htai-empty} == \text{hta-empty}

\textbf{definition} \ \text{htai-add-qi} :: - \Rightarrow \text{htai} \Rightarrow \text{htai}
\text{where} \ \text{htai-add-qi} == \text{hta-add-qi}
definition htau-add-rule :: - ⇒ htau ⇒ htau
  where htau-add-rule == hta-add-rule

definition htau-bwd-reduce :: htau ⇒ htau
  where htau-bwd-reduce == hta-bwd-reduce

definition htau-is-empty-witness :: htau ⇒ -
  where htau-is-empty-witness == hta-is-empty-witness

definition htau-ensure-idx-f :: htau ⇒ htau
  where htau-ensure-idx-f == hta-ensure-idx-f

definition htau-ensure-idx-s :: htau ⇒ htau
  where htau-ensure-idx-s == hta-ensure-idx-s

definition htau-ensure-idx-sf :: htau ⇒ htau
  where htau-ensure-idx-sf == hta-ensure-idx-sf

definition htaip-prod :: htau ⇒ htau ⇒ (nat * nat, nat) hashedTa
  where htaip-prod == hta-prod

definition htaip-prodWR :: htau ⇒ htau ⇒ (nat * nat, nat) hashedTa
  where htaip-prodWR == hta-prodWR

definition htaip-reindex :: (nat * nat,nat) hashedTa ⇒ htau
  where htaip-reindex == hta-reindex

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

    lemma htau-empty-correct[simp]:
      hta-α (htau-empty ()) = ta-empty
      hashedTa (htau-empty ())
      ⟨proof⟩

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

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

    lemmas htau-ensure-idx-f-correct =
      htau-ensure-idx-f-correct[folded htau-ensure-idx-f-def]
    lemmas htau-ensure-idx-s-correct =
      htau-ensure-idx-s-correct[folded htau-ensure-idx-s-def]
    lemmas htau-ensure-idx-sf-correct =
      htau-ensure-idx-sf-correct[folded htau-ensure-idx-sf-def]

  end

lemma htau-prod-correct:
  assumes [simp]: hashedTa H1  hashedTa H2
shows
\( \text{ta-lang} (\text{hta-\(\alpha\)} (\text{htai-prod} H1 H2)) = \text{ta-lang} (\text{hta-}\alpha) H1 \cap \text{ta-lang} (\text{hta-}\alpha H2) \)
\( \text{hashedTa} (\text{hta-prod} H1 H2) \)
(proof)

lemma hta-prodWR-correct:
assumes [simp]: \(\text{hashedTa} H1 \text{ hashedTa} H2\)
shows
\( \text{ta-lang} (\text{hta-}\alpha (\text{htai-prodWR} H1 H2)) = \text{ta-lang} (\text{hta-}\alpha H1) \cap \text{ta-lang} (\text{hta-}\alpha H2) \)
\( \text{hashedTa} (\text{hta-prodWR} H1 H2) \)
(proof)

lemma hta-union-correct:
assumes [simp]: \(\text{hashedTa} H1 \text{ hashedTa} H2\)
shows
\( \text{ta-lang} (\text{hta-}\alpha (\text{htai-union} H1 H2)) = \text{ta-lang} (\text{hta-}\alpha H1) \cup \text{ta-lang} (\text{hta-}\alpha H2) \)
\( \text{hashedTa} (\text{hta-union} H1 H2) \)
(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 \text{htai} instead of \text{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

\( \text{hta-empty-correct}: \text{hta-}\(\alpha\) (\text{hta-empty} ()) = \text{ta-empty} \)
\( \text{hashedTa} (\text{hta-empty} ()) \)
\( \text{ta-empty-lang}: \text{ta-lang} \text{ta-empty} = {} \)
**Function:** `hta-add-qi`
Adds an initial state to the given automaton.

**Relevant Lemmas**

\[
\text{hashedTa}.hta-add-qi-correct \quad \text{hashedTa} \ H \implies hta-\alpha \ (hta-add-qi \ qi \ H) = \\
(\langle ta-initial = \text{insert qi (ta-initial (hta-\alpha \ H)), ta-rules = ta-rules (hta-\alpha \ H) \rangle) \\
\text{hashedTa} \ H \implies \text{hashedTa} \ (hta-add-qi \ qi \ H)
\]

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

**Relevant Lemmas**

\[
\text{hashedTa}.hta-add-rule-correct \quad \text{hashedTa} \ H \implies hta-\alpha \ (hta-add-rule \ r \ H) = \\
(\langle ta-initial = ta-initial (hta-\alpha \ H), ta-rules = \text{insert r (ta-rules (hta-\alpha \ H))} \rangle) \\
\text{hashedTa} \ H \implies \text{hashedTa} \ (hta-add-rule \ r \ H)
\]

### 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**

\[
\text{hashedTa}.hta-mem-correct \quad \text{hashedTa} \ H \implies hta-mem \ t \ H = (t \in \text{ta-lang} (hta-\alpha \ H))
\]

\[
\text{hashedTa}.hta-mem'-correct \quad [\text{hashedTa} \ H; hta-has-idx-f \ H] \implies hta-mem' \\
t \ H = (t \in \text{ta-lang} (hta-\alpha \ H))
\]

66
**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: $[[\text{hashedTa } H1; \text{hashedTa } H2]] \implies $hta-α $(hta-prod$ $H1 H2)$
$hta-prod$ $H1 H2)$

$hta-prod$-correct: $[[\text{hashedTa } H1; \text{hashedTa } H2]] \implies $ta-lang $(hta-α $(hta-prod

$H1 H2)$) $\cap$ ta-lang $(hta-α H2)$

$hta-prod$-correct-aux: $[[\text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2]] \implies $hta-α $(hta-prod$ $H1 H2)$

$hta-prod$-correct: $[[\text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2]] \implies $hta-α $(hta-prod$ $H1 H2)$

$hta-prod$-correct-aux: $[[\text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2]] \implies $hta-α $(hta-prod$ $H1 H2)$

$hta-prod$-correct: $[[\text{hashedTa } H1; \text{hashedTa } H2; \text{hta-has-idx-s } H1; \text{hta-has-idx-sf } H2]] \implies $hta-α $(hta-prod$ $H1 H2)$

**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: $[[\text{hashedTa } H1; \text{hashedTa } H2]] \implies $hta-α $(hta-prodWR$ $H1 H2)$

$hta-prodWR$-correct: $[[\text{hashedTa } H1; \text{hashedTa } H2]] \implies $hta-α $(hta-prodWR$ $H1 H2)$

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

hta-union-correct: \[
\text{[hashedTa } H_1; \text{ hashedTa } H_2] \implies \text{ hta-} \alpha (\text{ hta-union } H_1 \ H_2) = \text{ ta-union-wrap } (\text{ hta-} \alpha H_1) (\text{ hta-} \alpha H_2)
\]
\[
\text{[hashedTa } H_1; \text{ hashedTa } H_2] \implies \text{ hashedTa } (\text{ hta-union } H_1 \ H_2)
\]

hta-union-correct: \[
\text{[hashedTa } H_1; \text{ hashedTa } H_2] \implies \text{ ta-lang } (\text{ hta-} \alpha (\text{ hta-union } H_1 \ H_2)) = \text{ ta-lang } (\text{ hta-} \alpha H_1) \cup \text{ ta-lang } (\text{ hta-} \alpha H_2)
\]
\[
\text{[hashedTa } H_1; \text{ hashedTa } H_2] \implies \text{ hashedTa } (\text{ hta-union } H_1 \ H_2)
\]

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

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

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

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

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
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 it's axiomatization in Isabelle is inconsistent and no one found and reported this inconsistency already.

The next crucial point is the code generator of Isabelle. We derive executable code from our specifications. The code generator contains another (thin) layer of untrusted code. This layer has some known deficiencies\(^2\) (as of Isabelle2009) in the sense that invalid code is generated. This code is then rejected by the target language’s compiler or interpreter, but does not silently compute the wrong thing.

Moreover, assuming correctness of the code generator, the generated code is only guaranteed to be partially correct\(^3\), i.e. there are no formal termination guarantees.

\(^2\)For example, the Haskell code generator may generate variables starting with uppercase letters, while the Haskell-specification requires variables to start with lowercase letters. Moreover, the ML code generator does not know the ML value restriction, and may generate code that violates this restriction.

\(^3\)A simple example is the always-diverging function \(f_{div} : \text{bool} \rightarrow \text{while} (\lambda x. \text{True}) \text{id} \text{True}\) that is definable in HOL. The lemma \(\forall x. x = \text{if } f_{div} \text{ then } x \text{ else } x\) is provable in Isabelle and rewriting based on it could, theoretically, be inserted before the code generation process, resulting in code that always diverges.
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.
References


