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 functions of the tree automata interface.

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

Main.ml Main file for compiled performance tests.

Main_script.ml Main file for scripted performance tests.

make.sh Compile performance test files.

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

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

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

1.1.7 code/haskell/

This directory contains the Haskell code.

code/haskell/generated/ Contains the files generated by Isabelle’s code generator. The Ta.hs declares the module Ta that contains the tree automata interface. There may be more files in this directory, that declare modules that are imported by Ta.
doTests.sh Compile and execute performance tests.

Main.hs Source-code of performance tests.

make.sh Compile performance tests.

Pt_examples.hs Input data for performance tests.

1.1.8 code/taml/

This directory contains the Timbuk/Taml test cases.

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

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

2 Trees

theory Tree
imports Main
begin

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

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

datatype-compat tree

end

3 Tree Automata

theory Ta
imports Main ../Automatic-Refinement/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 ( - → - )

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

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

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

3.1.2 Acceptance

The predicate \( \text{accs} \ δ \ t \ q \) 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⟩

9
lemma δ-states-accsI: accs δ n q =⇒ q ∈ δ-states δ
(proof)

lemma δ-states-union[simp]: δ-states (δ ∪ δ′) = δ-states δ ∪ δ-states δ′
(proof)

lemma δ-states-insert[simp]:
d-states (insert r δ) = (rule-states r ∪ δ-states δ)
(proof)

lemma δ-states-mono: [δ ⊆ δ′] =⇒ δ-states δ ⊆ δ-states δ′
(proof)

lemma δ-states-finite[simp, intro]: finite δ =⇒ finite (δ-states δ)
(proof)

lemma δ-statesE: [q ∈ δ-states ∆;
!f qs. [(q → f qs) ∈ ∆] =⇒ P;
!ql f qs. [(ql → f qs) ∈ ∆; q ∈ set qs] =⇒ P
] =⇒ P
(proof)

lemma δ-symbolsI: (q → f qs) ∈ δ =⇒ f ∈ δ-symbols δ
(proof)

lemma δ-symbolsE:
assumes A: f ∈ δ-symbols δ
obtains q qs where (q → f qs) ∈ δ
(proof)

lemma δ-symbols-simps[simp]:
d-symbols { } = { }
d-symbols (insert r δ) = insert (rhs1 r) (δ-symbols δ)
d-symbols (δ ∪ δ′) = δ-symbols δ ∪ δ-symbols δ′
(proof)

lemma δ-symbols-finite[simp, intro]:
finite δ =⇒ finite (δ-symbols δ)
(proof)

lemma accs-mono: [accs δ n q; δ ⊆ δ′] =⇒ accs δ′ n q
(proof)

context tree-automaton
begin
lemma initial-subset: ta-initial TA ⊆ ta-rstates TA
(proof)
lemma states-subset: δ-states (ta-rules TA) ⊆ ta-rstates TA
(proof)
lemma finite-states[simp, intro!]: finite (ta-rstates TA)
(proof)

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

lemmas is-subset = 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
— Finite tree automata with ranked alphabet

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

begin

lemma rules-legal: r∈δ \implies r∈legal-rules Q
(proof)
lemma accs-is-ranked: accs δ t q \implies 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 \implies q = q'

begin

theorem accs-unique: accs δ t q; accs δ t q' \implies q = q'
(proof)
end

3.3.3 Complete Tree Automata

locale complete-tree-automaton = det-tree-automaton TA A
for TA :: ('Q', 'L) tree-automaton-rec and A +
assumes complete:
qs∈lists Q; A f = Some (length qs) \implies \exists 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 \implies \exists q∈Q. accs δ t q
(proof)
end

12
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 \( q \rightarrow l \) qs \( = (f \ q \rightarrow l \ \langle \text{map f qs} \rangle) \)

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\in ta-lang TA => t\in ta-lang (ta-remap f TA)
  (proof)

lemma remap-accs2: [ 
  accs δ' n q';
  δ'=\(\text{remap-rule f \ ' δ} \);
  q'=f q;
  inj-on f Q;
  q\in Q;
  δ-states δ \subseteq 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 δn 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
\(\text{proof}\)

**corollary** accs-exclusive2:
\[
\begin{align*}
\text{accs}\left(\delta \cup \delta'\right) \cap \text{\delta-states} \cap \text{\delta'-states} = \{\};
\end{align*}
\]
\(\implies\) accs \(\delta' n q\)
\(\text{proof}\)

**lemma** ta-union-correct-aux1:
\begin{align*}
&\text{fixes} TA, TA' \\
&\text{assumes} TA: \text{tree-automaton TA} \\
&\text{assumes} TA': \text{tree-automaton TA'} \\
&\text{assumes DJ: ta-rstates } TA \cap \text{ta-rstates } TA' = \{\} \\
&\text{shows} \text{ta-lang } (\text{ta-union } TA \cup \text{ta-lang } TA')
\end{align*}
\(\text{proof}\)

**lemma** ta-union-correct-aux2:
\begin{align*}
&\text{fixes} TA, TA' \\
&\text{assumes} TA: \text{tree-automaton TA} \\
&\text{assumes} TA': \text{tree-automaton TA'} \\
&\text{shows} \text{tree-automaton } (\text{ta-union } TA \cup \text{ta-lang } TA')
\end{align*}
\(\text{proof}\)

**theorem** ta-union-correct:
\begin{align*}
&\text{fixes} TA, TA' \\
&\text{assumes} TA: \text{tree-automaton TA} \\
&\text{assumes} TA': \text{tree-automaton TA'} \\
&\text{assumes DJ: ta-rstates } TA \cap \text{ta-rstates } TA' = \{\} \\
&\text{shows} \text{ta-lang } (\text{ta-union } TA \cup \text{ta-lang } TA') \\
&\text{tree-automaton } (\text{ta-union } TA \cup \text{ta-lang } TA')
\end{align*}
\(\text{proof}\)

**lemma** ta-union-rta:
\begin{align*}
&\text{fixes} TA, TA' \\
&\text{assumes} TA: \text{ranked-tree-automaton } TA A \\
&\text{assumes} TA': \text{ranked-tree-automaton } TA' A \\
&\text{shows} \text{ranked-tree-automaton } (\text{ta-union } TA \cup \text{ta-lang } TA') A
\end{align*}
\(\text{proof}\)

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

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

**lemma** usw-disjoint[simp]:
\begin{align*}
&USW1 \cdot X \cap USW2 \cdot Y = \{} \\
&\text{remap-rule } USW1 \cdot X \cap \text{remap-rule } USW2 \cdot Y = \{}
\end{align*}
\(\text{proof}\)

**lemma** states-usw-disjoint[simp]:
\begin{align*}
&\text{ta-rstates } (\text{ta-remap USW1 X}) \cap \text{ta-rstates } (\text{ta-remap USW2 Y}) = \{}
\end{align*}
\(\text{proof}\)
lemma usw-inj-on[simp, intro!]:
inj-on USW1 X
inj-on USW2 X
⟨proof⟩
definition ta-union-wrap TA TA’ =
ta-union (ta-remap USW1 TA) (ta-remap USW2 TA’)
lemma ta-union-wrap-correct:
fixes TA :: (’Q1,’L) tree-automaton-rec
fixes TA’ :: (’Q2,’L) tree-automaton-rec
assumes TA: tree-automaton TA
assumes TA’: tree-automaton TA’
shows ta-lang (ta-union-wrap TA TA’) = ta-lang TA ∪ ta-lang TA’ (is ?T1)
tree-automaton (ta-union-wrap TA TA’) (is ?T2)
⟨proof⟩
lemma ta-union-wrap-rta:
fixes TA TA’
assumes TA: ranked-tree-automaton TA A
assumes TA’: ranked-tree-automaton TA’ A
shows ranked-tree-automaton (ta-union-wrap TA TA’) A
⟨proof⟩

3.4.4 Reduction
definition reduce-rules δ P == δ ∩ { r. rule-states r ⊆ P }
lemma reduce-rulesI: [] r∈δ; rule-states r ⊆ P [] r∈reduce-rules δ P
⟨proof⟩
lemma reduce-rulesD:
[ r∈reduce-rules δ P ] r∈δ
[ r∈reduce-rules δ P; q∈rule-states r ] q∈P
⟨proof⟩
lemma reduce-rules-subset: reduce-rules δ P ⊆ δ
⟨proof⟩
lemma reduce-rules-mono: P ⊆ P’ ⇒ reduce-rules δ P ⊆ reduce-rules δ P’
⟨proof⟩
lemma δ-states-reduce-subset:
shows δ-states (reduce-rules δ Q) ⊆ δ-states δ ∩ Q
⟨proof⟩
lemmas δ-states-reduce-subsetI = set-rev-mp[OF - δ-states-reduce-subset]
definition \textit{ta-reduce}
\[ \text{\texttt{\textit{ta-reduce}}:: ('}Q',L') \Rightarrow ('}Q',L') \Rightarrow ('}Q',L') \text{\texttt{\textit{tree-automaton-rec}}} \]
\textbf{where} \textit{ta-reduce} TA P ==
\[ (\mid \text{\textit{ta-initial}} = \text{\textit{ta-initial}} TA \cap P, \]
\[ \text{\textit{ta-rules}} = \text{\textit{reduce-rules}} (\text{\textit{ta-rules}} TA) P \] \]

Reducing a tree automaton preserves the tree automata invariants

\textbf{Theorem} \textit{ta-reduce-inv}: \textbf{Assumes} A: \textit{tree-automaton} TA
\textbf{shows} \textit{tree-automaton} (\textit{ta-reduce} TA P)

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

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

\textbf{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 \(q \leftarrow f(q,\ldots)\).

\textbf{— Forward successors}

\textbf{Inductive-set} \textit{f-succ} for \(\delta\) where
\[ [(q \rightarrow l qs)\in\delta; q'\in set qs] \Rightarrow (q,q') \in f\text{-succ} \delta \]

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

\textbf{Definition} \textit{f-accessible} \(\delta\) \(Q^0\) == \((f\text{-succ} \delta)^*\) \(\Rightarrow\) \(Q^0\)

\textbf{— 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.}

\textbf{Inductive-set} \textit{f-accessible-alt} :: \(('}Q',L')\) \textit{ta-rule set} \Rightarrow 'Q set \Rightarrow 'Q set
\textbf{for} \(\delta\) \(Q^0\)
\textbf{where}
\[ \textit{fa-refl}: q^0\in Q^0 \Rightarrow q^0 \in f\text{-accessible-alt}\ \delta\ \ Q^0 \] \[ \textit{fa-step}: [ q\in f\text{-accessible-alt}\ \delta\ \ Q^0; (q \rightarrow l qs)\in\delta; q'\in set qs ] \]
\[ \Rightarrow q' \in f\text{-accessible-alt}\ \delta\ \ Q^0 \]
\textbf{Lemma} \textit{f-accessible-alt}: \textit{f-accessible} \(\delta\) \(Q^0\) = \textit{f-accessible-alt} \(\delta\) \(Q^0\)
\textbf{(proof)}

\textbf{Lemmas} \textit{f-accessibleI} = \textit{f-accessible-alt}.\textit{intros}[folded \textit{f-accessible-alt}]
\textbf{Lemmas} \textit{f-accessibleE} = \textit{f-accessible-alt}.\textit{cases}[folded \textit{f-accessible-alt}]

17
lemma f-succ-finite[simp, intro]: finite δ ⇒ finite (f-succ δ)
(proof)

lemma f-accessible-mono: Q ⊆ Q′ ⇒ x ∈ f-accessible δ Q ⇒ x ∈ f-accessible δ Q′
(proof)

lemma f-accessible-prepend:
[( q → l qs ) ∈ δ; q′ ∈ set qs; x ∈ f-accessible δ { q′ }] ⇒ x ∈ f-accessible δ { q}
(proof)

lemma f-accessible-subset: q ∈ f-accessible δ Q ⇒ q ∈ Q ∪ δ-states δ
(proof)

lemma (in tree-automaton) f-accessible-in-states:
q ∈ f-accessible (ta-rules TA) (ta-initial TA) ⇒ q ∈ ta-rstates TA
(proof)

lemma f-accessible-refl-inter-simp[simp]: Q ∩ f-accessible r Q = Q
(proof)

lemma accs-reduce-f-acc[simp]: accs δ t q =⇒ accs (reduce-rules δ (f-accessible δ { q})) t q
(proof)

abbreviation ta-fwd-reduce TA ==
(ta-reduce TA (f-accessible (ta-rules TA) (ta-initial TA)))

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

theorem ta-reduce-f-acc[simp]: ta-lang (ta-fwd-reduce TA) = ta-lang TA
(proof)

**Backward Reduction**  A state is backward accessible, iff at least one tree is accepted in it.
Inductively, backward accessible states can be characterized as follows: A state is backward accessible, if it occurs on the left hand side of a rule, and all states on this rule’s right hand side are backward accessible.

inductive-set b-accessible :: ('Q,'L) ta-rule set ⇒ 'Q set
for δ
where
[( q → l qs ) ∈ δ; ∀x. x ∈ set qs ⇒ x ∈ b-accessible δ ] ⇒ q ∈ b-accessible δ

lemma b-accessibleI:
[( q → l qs ) ∈ δ; set qs ⊆ b-accessible δ ] ⇒ q ∈ b-accessible δ
(proof)

lemma accs-is-b-accessible: accs δ t q ⇒ q ∈ b-accessible δ
(proof)

18
**Lemma** b-acc-subset-δ-statesI: \( x \in b\text{-accessible } \delta \implies x \in \delta\text{-states } \delta \)

\((\text{proof})\)

**Lemma** b-acc-subset-δ-states: \( b\text{-accessible } \delta \subseteq \delta\text{-states } \delta \)

\((\text{proof})\)

**Lemma** b-acc-finite[simp, intro!]: finite \( \delta \implies \) finite (b-accessible \( \delta \))

\((\text{proof})\)

**Lemma** b-accessible-is-accs:

\[ q \in b\text{-accessible } \text{(ta-rules } TA \text{)};\]

\[ \exists ! l. \text{ accs } \text{(ta-rules } TA \text{)} t q \implies P \]

\((\text{proof})\)

**Lemma** accs-reduce-b-acc:

\[ \text{acco } \delta t q \implies \text{accs } \text{(reduce-rules } \delta \text{ (b-accessible } \delta \text{)) } t q \]

\((\text{proof})\)

**Abbreviation** ta-bwd-reduce TA == (ta-reduce TA (b-accessible (ta-rules TA)))

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

**Theorem** ta-reduce-b-acc[simp]: ta-lang (ta-bwd-reduce TA) = ta-lang TA

\((\text{proof})\)

**Theorem** empty-if-no-b-accessible:

\[ \text{ta-lang } TA = \{ \} \iff \text{ta-initial } TA \cap b\text{-accessible } \text{(ta-rules } TA \text{)} = \{ \} \]

\((\text{proof})\)

### 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

\[ r\text{-prod } (q1 \rightarrow l q1) (q2 \rightarrow l q2) = ((q1,q2) \rightarrow l l (\text{zip } q1 q2)) \]

— Product rules

**Definition** δ-prod δ1 δ2 ==

\[
\{ \\
\text{r-prod } (q1 \rightarrow l q1) (q2 \rightarrow l q2) \mid q1 q2 l q1 q2.} \\
\text{length } q1 = \text{length } q2 \land \\
(q1 \rightarrow l q1) \in \delta1 \land \\
(q2 \rightarrow l q2) \in \delta2 \\
\}
\]

**Lemma** δ-prodI:

\[
\begin{align*}
\text{length } q1 &= \text{length } q2; \\
(q1 \rightarrow l q1) &\in \delta1; \\
(q2 \rightarrow l q2) &\in \delta2 \implies ((q1,q2) \rightarrow l (\text{zip } q1 q2)) \in \delta\text{-prod } \delta1 \delta2
\end{align*}
\]

\((\text{proof})\)

**Lemma** δ-prodE:

\[
\]
\( r \in \delta \prod \delta_1 \delta_2; \)
\(] \quad q_1 q_2 l qs_1 qs_2. [\ length qs_1 = length qs_2; \)
\( (q_1 \rightarrow l \ qs_1) \in \delta_1; \)
\( (q_2 \rightarrow l \ qs_2) \in \delta_2; \)
\( r = (q_1, q_2) \rightarrow l (zip \ qs_1 \ qs_2)) \]
\]
\(] \quad \Rightarrow \quad P \)

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

\textbf{lemma } \( \delta \prod \text{-precise}: \)
\(] \quad accs \ \delta_1 t q_1; \ accs \ \delta_2 t q_2 \] \( \Rightarrow \ accs \ (\delta \prod \delta_1 \delta_2) t (q_1, q_2) \)
\textbf{⟨proof⟩}

\textbf{lemma } \( \delta \prod \text{-empty}[\text{simp}]: \)
\( \delta \prod \{\} \ \delta = \{\} \)
\textbf{⟨proof⟩}

\textbf{lemma } \( \delta \prod \text{-2sg}[\text{simp}]: \)
\(] \quad rhsl r_1 \neq rhsl r_2 \] \( \Rightarrow \ \delta \prod \{r_1\} \ \{r_2\} = \{\} \)
\(] \quad length (rhql r_1) \neq length (rhql r_2) \] \( \Rightarrow \ \delta \prod \{r_1\} \ \{r_2\} = \{\} \)
\(] \quad rhsl r_1 = rhsl r_2; \ length (rhql r_1) = length (rhql r_2) \]
\( \Rightarrow \ \delta \prod \{r_1\} \ \{r_2\} = \{r \prod r_1 \ r_2\} \)
\textbf{⟨proof⟩}

\textbf{lemma } \( \delta \prod \text{-Un}[\text{simp}]: \)
\( \delta \prod (\delta_1 \cup \delta_1') \delta_2 = \delta \prod \delta_1 \delta_2 \cup \delta \prod \delta_1' \delta_2 \)
\( \delta \prod \delta_1 (\delta_2 \cup \delta_2') = \delta \prod \delta_1 \delta_2 \cup \delta \prod \delta_1 \delta_2' \)
\textbf{⟨proof⟩}

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

\textbf{definition } \( \delta \prod \text{-sng1 } r \ \delta_2 = \)
\textbf{case } \( r \) of \( (q_1 \rightarrow l \ qs_1) \) ⇒
\( \{ r \prod \ (q_2 \rightarrow l \ qs_2) | \)
\( q_2 \ qs_2, \ \length \ qs_1 = \length \ qs_2 \ \land \ (q_2 \rightarrow l \ qs_2) \in \delta_2 \)
\}

\textbf{definition } \( \delta \prod \text{-sng2 } \delta_1 \ r = \)
\textbf{case } \( r \) of \( (q_2 \rightarrow l \ qs_2) \) ⇒
\( \{ r \prod \ (q_1 \rightarrow l \ qs_1) \ r | \)
\( q_1 \ qs_1, \ \length \ qs_1 = \length \ qs_2 \ \land \ (q_1 \rightarrow l \ qs_1) \in \delta_1 \)
\}

\textbf{lemma } \( \delta \prod \text{-sng-alt}: \)
\( \delta \ prod \text{-sng1 } r \ \delta_2 = \delta \ prod \ \{r\} \ \delta_2 \)

\[ 20 \]
δ-prod-sng2 δ1 r = δ-prod δ1 {r}
⟨proof⟩

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

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

lemma ta-prod-correct-aux1:
ta-lang (ta-prod TA1 TA2) = ta-lang TA1 ∩ ta-lang TA2
⟨proof⟩

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

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

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

theorem ta-prod-correct:
assumes TA: tree-automaton TA1 tree-automaton TA2
shows
ta-lang (ta-prod TA1 TA2) = ta-lang TA1 ∩ ta-lang TA2
  tree-automaton (ta-prod TA1 TA2)
⟨proof⟩

lemma ta-prod-rta:
assumes TA: ranked-tree-automaton TA1 A ranked-tree-automaton TA2 A
shows ranked-tree-automaton (ta-prod TA1 TA2) A
⟨proof⟩

3.4.6 Determinization

We only formalize the brute-force subset construction without reduction.
The basic idea of this construction is to construct an automaton where the
states are sets of original states, and the lhs of a rule consists of all states
that a term with given rhs and function symbol may be labeled by.
context ranked-tree-automaton
begin
  — Left-hand side of subset rule for given symbol and rhs
definition δss-lhs f ss ==
  \{ q | q qs. (q \rightarrow f qs) \in \delta \land \text{list-all-zip (op \in) qs ss} \}

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

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

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

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

lemma δss-det: \[ \[(q \rightarrow f qs) \in \deltass; (q' \rightarrow f qs) \in \deltass \] \implies q = q' \]
  ⟨proof⟩

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

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

definition detTA == ( \{ ta-initial = \{ s. s \subseteq Q \land s \cap Qi \neq \{ \} \},
  ta-rules = \deltass \})

theorem detTA-is-ta[simp, intro]:
  det-tree-automaton detTA A
  ⟨proof⟩
\textbf{Theorem} \textit{detTA-lang[simp]}:
\begin{align*}
\text{ta-lang} (\text{detTA}) &= \text{ta-lang} \ TA \\
\langle \text{proof} \rangle
\end{align*}

\textbf{Theorems} \textit{detTA-correct} = \textit{detTA-is-ta} \textit{detTA-lang}
end

\textbf{3.4.7 Completion}

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

\textbf{Context} \textit{det-tree-automaton}

\begin{quote}
\textbf{Definition} \textit{Qcomplete} == \textit{insert None} (\text{Some'}Q)
\end{quote}

\textbf{Lemma} \textit{Qcomplete-finite[simp, intro]}: \textit{finite} \textit{Qcomplete}
\langle \text{proof} \rangle

\textbf{Definition} \textit{\delta}complete :: ('Q option, 'L) \textit{ta-rule set where}
\begin{align*}
\delta\text{complete} &= (\text{remap-rule} \ \text{Some} ' \delta) \\
& \cup \{ (\text{None} \rightarrow f \ \text{qs}) | f \ \text{qs}, \ A \ f = \text{Some} (\text{length} \ \text{qs}) \\
& \land \ \text{qs} \in \text{lists} \ \text{Qcomplete} \\
& \land \neg(\exists \ qo \ qso. \ (qo \rightarrow f \ qso) \in \delta \land \ \text{qs}=\text{map} \ \text{Some} \ qso) \}
\end{align*}

\textbf{Lemma} \textit{\delta-states-complete}: \textit{q} \in \textit{\delta-states} \ \textit{\delta}complete \implies \textit{q} \in \textit{Qcomplete}
\langle \text{proof} \rangle

\textbf{Definition}
\begin{align*}
\text{completeTA} &= \{ \ \text{ta-initial} = \text{Some'}Qi, \ \text{ta-rules} = \delta\text{complete} \ |
\end{align*}

\textbf{Lemma} \textit{\delta-complete-finite[simp, intro]}: \textit{finite} \textit{\delta}complete
\langle \text{proof} \rangle

\textbf{Theorem} \textit{completeTA-is-ta}: \textit{complete-tree-automaton completeTA} \ A
\langle \text{proof} \rangle

\textbf{Theorem} \textit{completeTA-lang}: \textit{ta-lang completeTA} = \textit{ta-lang} \ TA
\langle \text{proof} \rangle

\textbf{Theorems} \textit{completeTA-correct} = \textit{completeTA-is-ta} \textit{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 = \( Q - Q_i \),
ta-rules = \( \delta \ )

lemma cta-rules[simp]: ta-rules complementTA = \( \delta \ )
(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 :: (\( \text{nat} \to \text{nat} \)) \to \text{nat tree set set}
where regular-languages A ==
  \{ ta-lang TA \mid (TA::(\text{nat,\text{nat}}) tree-automaton-rec).
  ranked-tree-automaton TA A \}

lemma rtlE:
fixes L :: \text{nat tree set}
assumes A: L \in regular-languages A
obtains TA::(nat,\text{nat}) 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 \in 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{align*}
\text{obtains } TAC &:: (Q \text{ set option}, L) \text{ tree-automaton-rec where} \\
\text{ta-lang } TAC & = \text{ta-lang } TA \\
\text{complete-tree-automaton } TAC & = A \\
\end{align*}

\textbf{end}

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

\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{theorem} rtl-union-closed:
\begin{align*}
\begin{bmatrix}
L1 &\in & \text{regular-languages } A; \\
L2 &\in & \text{regular-languages } A
\end{bmatrix}
\Rightarrow 
L1 \cup L2 &\in \text{regular-languages } A
\end{align*}

\textbf{theorem} rtl-inter-closed:
\begin{align*}
\begin{bmatrix}
L1 &\in & \text{regular-languages } A; \\
L2 &\in & \text{regular-languages } A
\end{bmatrix}
\Rightarrow 
L1 \cap L2 &\in \text{regular-languages } A
\end{align*}

\textbf{theorem} rtl-complement-closed:
\begin{align*}
L &\in \text{regular-languages } A 
\Rightarrow 
\text{ranked-trees } A - L &\in \text{regular-languages } A
\end{align*}

\textbf{theorem} (in finite-alphabet) rtl-univ:
\begin{align*}
\text{ranked-trees } A &\in \text{regular-languages } A
\end{align*}
proof

\textbf{Theorem rtl-diff-closed:}
\begin{itemize}
\item \textbf{Fixes} \( L1 \) :: 'L tree set
\item \textbf{Assumes} \( \text{A[simp]}: L1 \in \text{regular-languages} A \quad L2 \in \text{regular-languages} A \)
\item \textbf{Shows} \( L1 - L2 \in \text{regular-languages} A \)
\end{itemize}
\textbf{Proof} \quad

\textbf{Theorems rtl-closed = finite-alphabet.rtl-empty finite-alphabet.rtl-univ}
\textbf{rtl-complement-closed}
\textbf{rtl-inter-closed rtl-union-closed rtl-diff-closed}

\textbf{End}

\section{Abstract Tree Automata Algorithms}

\textbf{Theory AbsAlgo}

\textbf{Imports}
\begin{itemize}
\item Ta
\item ../Collections/Examples/ICF/Exploration
\item ../Collections/ICF/CollectionsV1
\end{itemize}

\textbf{Begin}

\textbf{No-Notation} fun-rel-syn (infixr \( \rightarrow \) 60)

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

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

\subsection{Word Problem}

First, a recursive version of the \texttt{accs}-predicate is defined.

\begin{verbatim}
fun r-match :: 'a set list \Rightarrow 'a list \Rightarrow bool where
  r-match [] [] \iff True |
  r-match (A#AS) (a#as) \iff a\in A \land r-match AS as |
  r-match _ _ \iff 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 \iff length L = length l \land \forall i < length l. \exists i \in L. l!

(proof)

fun r-matchc where
  r-matchc q l Qs (qr \rightarrow lr qsr) \iff q=qr \land l=lr \land r-match Qs qsr
\end{verbatim}
— recursive version of accs-predicate

fun faccs :: ('Q,'L) ta-rule set => 'L tree => 'Q set where
faccs δ (NODE f ts) = 
  let Qs = map (faccs δ) (ts) in
  \{ q. \exists r∈δ. r-matchc q f Qs r \}

lemma faccs-correct-aux:
  q∈faccs δ n = accs δ n q (is ?T1)
  (map (faccs δ) ts = map (λ t. { q . accs δ t q}) ts) (is ?T2)
⟨proof⟩

theorem faccs-correct1: q∈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:

\(\alpha\) On this level, the state consists of a set of discovered states and a workset.

\(\alpha'\) 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.

\(\alpha\) - Level: — A state contains the set of discovered states and a workset

**Type-synonym** \(\langle Q', L \rangle_{br-state} = \langle Q set \rangle \times \langle Q set \rangle\)

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

**Definition** \(br-dsq\)

\[\begin{align*}
\langle Q', L \rangle_{ta-rule set} & \Rightarrow \langle Q' \rangle \Rightarrow \langle Q, L \rangle_{br-state} \Rightarrow \langle Q set \rangle \\
where \quad br-dsq \delta q &= \lambda(Q,W). \{ \text{lhs } r \mid r. \ r\in\delta \land \text{set } (\text{rhsq } r) \subseteq (Q-(W-\{q\}))\} \\
\end{align*}\]

— 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** \(br-step\)

\[\langle (Q', L)_{ta-rule set} \Rightarrow \langle (Q', L)_{br-state} \times (Q', L)_{br-state} \rangle \rangle \text{ set} \]

**For** \(\delta\) where

\[\begin{align*}
\quad q\in W; \\
\quad Q' = Q \cup br-dsq \delta q (Q,W); \\
\quad W' = W - \{q\} \cup (br-dsq \delta q (Q,W) - Q) \\
\quad \Rightarrow ((Q,W),(Q',W')) \in br-step \delta \\
\end{align*}\]

— Termination condition for backwards reduction: The workset is empty

**Definition** \(br-cond\)

\[\langle (Q', L)_{ta-rule set} \Rightarrow \langle (Q, L)_{br-state} \rangle \text{ set} \]

**Where** \(br-cond \quad \Rightarrow \{((Q,W), W\neq\{\})\} \)

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

**Definition** \(bre-cond\)

\[\langle Q' \rangle_{br-state} \Rightarrow \langle (Q', L)_{br-state} \rangle \text{ set} \]

**Where** \(bre-cond \quad Qi \Rightarrow \{((Q,W), W\neq\{\} \land (Qi\cap Q=\{\})\} \)

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

**Definition** \(br-iq\)

\[\langle (Q', L)_{ta-rule set} \Rightarrow \langle Q set \rangle \text{ set} \]

**Where** \(br-iq \quad \Rightarrow \{ \text{lhs } r \mid r. \ r\in\delta \land \text{rhsq } r = \[] \} \)

— Initial state for the iteration

**Definition** \(br-initial\)

\[\langle (Q', L)_{ta-rule set} \Rightarrow \langle (Q', L)_{br-state} \rangle \text{ set} \]

**Where** \(br-initial \quad \Rightarrow \{(br-iq \delta, br-iq \delta)\} \)
— Invariant for the iteration:

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

**definition** br-invar :: ('Q, 'L) ta-rule set ⇒ ('Q, 'L) br-state set
**where** br-invar $\delta$ == {($Q, W$).
  $W \subseteq Q$ ∧ $Q \subseteq$ b-accessible $\delta$ ∧
bacc-step $\delta$ ($Q - W$) $\subseteq$ Q ∧
finite Q}

**definition** br-algo $\delta$ == ()
wa-cond = br-cond,
wa-step = br-step $\delta$,
wai-initial = {br-initial $\delta$},
wa-invar = br-invar $\delta$
)

**definition** bre-algo Qi $\delta$ == ()
wa-cond = bre-cond Qi,
wa-step = br-step $\delta$,
wai-initial = {br-initial $\delta$},
wa-invar = br-invar $\delta$
)

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

**definition** br-termrel $\delta$ ==
((($Q', Q$). $Q \subset Q'$ ∧ $Q' \subseteq$ b-accessible $\delta$)) <*lex*> finite-psubset

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

**lemma** br-termrel-uf[simp, intro!]: finite $\delta$ ⇒ uf {br-termrel $\delta$}
⟨proof⟩

**lemma** br-dsq-ss:
  assumes A: ($Q, W$)εbr-invar $\delta$ W ≠ {} qεW
  shows br-dsq $\delta$ q ($Q, W$) $\subseteq$ b-accessible $\delta$
⟨proof⟩

**lemma** br-step-in-termrel:
  assumes A: $\Sigma$εbr-cond $\Sigma$εbr-invar $\delta$ ($\Sigma, \Sigma'$)εbr-step $\delta$
shows $(\Sigma', \Sigma) \in \text{br-termrel } \delta$

(\text{proof})

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

(\text{proof})

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

(\text{proof})

lemma \text{br-invar-final}:
$\forall \Sigma. \Sigma \in \text{wa-invar } (\text{br-algo } \delta) \land \Sigma \notin \text{wa-cond } (\text{br-algo } \delta)$
$\longrightarrow \text{fst } \Sigma = \text{b-accessible } \delta$

(\text{proof})

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

(\text{proof})

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

(\text{proof})

theorem \text{bre-while-algo}:
assumes FIN[\text{simp}]: \text{finite } \delta
shows while-algo $(\text{bre-algo } Qi \delta)$

(\text{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).

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

— Abstraction to $\alpha$-level

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

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

30
definition \( br' \)-invar-add :: ('Q', 'L) ta-rule set \Rightarrow ('Q', 'L) br'-state set
where \( br' \)-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) = 0\} \subseteq Q \}

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

inductive-set \( br' \)-step
:: ('Q', 'L) ta-rule set \Rightarrow (('Q', 'L) br'-state \times ('Q', 'L) br'-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 \} 
\land 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 \br'\-step \ \delta

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

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

inductive-set \( br' \)-initial :: ('Q', 'L) ta-rule set \Rightarrow ('Q', 'L) br'-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 \br'\-initial \ \delta

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

lemma \( br' \)-step-invar:
assumes finite(simp): finite δ
assumes INV: Σ ∈ br′-invar-add δ  br′-α Σ ∈ br-invar δ
assumes STEP: (Σ, Σ′) ∈ br′-step δ
shows Σ′ ∈ br′-invar-add δ

⟨proof⟩

lemma br′-invar-initial:
br′-initial δ ⊆ br′-invar-add δ
⟨proof⟩

lemma br′-rcm-aux1:
assumes A: (Q, W, rcm) ∈ br′-invar δ  q ∈ W
shows {lhs r | r. r ∈ δ ∧ q ∈ set (rshq r) ∧ the (rcm r) ≤ Suc 0}
  = {lhs r | r. r ∈ δ ∧ q ∈ set (rshq r) ∧ set (rshq r) ⊆ (Q − (W − {q})))}
⟨proof⟩

lemma br′-rcm-aux2:
assumes A: (Q, W, rcm) ∈ br′-invar δ  q ∈ W
shows Q ⊇ br-dsq δ (Q, W)
  = Q ∪ {lhs r | r. r ∈ δ ∧ q ∈ set (rshq r) ∧ the (rcm r) ≤ Suc 0}
⟨proof⟩

lemma br′-rcm-aux3:
assumes A: (Q, W, rcm) ∈ br′-invar δ  q ∈ W
shows br-dsq δ q (Q, W) − Q
  = {lhs r | r. r ∈ δ ∧ q ∈ set (rshq r) ∧ the (rcm r) ≤ Suc 0} − Q
⟨proof⟩

lemma br′-step-abs:
Σ ∈ br′-invar δ:
(Σ, Σ′) ∈ br′-step δ
[⟨proof⟩

lemma br′-initial-abs: br′-α′(br′-initial δ) = {br-initial δ}
⟨proof⟩
lemma \( br'\)-cond-abs: \( \Sigma \in br'\)-cond \( \iff \) \( (br'\alpha \Sigma) \in br\)-cond

lemma \( bre'\)-cond-abs: \( \Sigma \in bre'\)-cond \( Qi \iff \) \( (br'\alpha \Sigma) \in bre\)-cond \( Qi\)

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

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

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

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

lemma \( fst-br'\alpha\): \( fst (br'\alpha s) = fst s\)

theorems \( br'\)-invar-final =
\( br'\)-pref.transfer-correctness[\( OF \) \( br\)-invar-final, unfolded \( \text{fst-br'\alpha}\)]

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

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

theorem \( bre'\)-while-algo:
finite \( \delta \implies \) while-algo \( (bre'\alpha Qi \delta)\)

theorems \( bre'\)-invar-final =
\( bre'\)-pref.transfer-correctness[\( OF \) \( bre\)-invar-final, unfolded \( \text{fst-br'\alpha}\)]

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
\( :: (Q,L) ta-rule \Rightarrow (Q,L) \) \( br'\)-state \Rightarrow (Q,L) \) \( br'\)-state
where
\( br'\)-inner-step \( = \lambda r \ (Q,W,rcm) \) let \( c=\text{the (rcm r) in } \)
\( \text{if } c \leq 1 \text{ then insert } (lhs r) \) \( Q \text{ else } Q, \)
\( \text{if } c \leq 1 \land (lhs r) \notin Q \text{ then insert } (lhs r) \) \( W \text{ else } W, \)
\( \text{rcm } (r \mapsto (c-(1::\text{nat})\)))\)
definition \( br'\text{-inner-invar} \)
\[
:: (Q', L) \text{ ta-rule set} \Rightarrow Q \Rightarrow (Q', L) \text{ br'\text{-state}}
\]
\[
\Rightarrow (Q', L) \text{ ta-rule set} \Rightarrow (Q', L) \text{ br'\text{-state}} \Rightarrow \text{bool}
\]

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

lemma \( br'\text{-inner-invar-imp-final} \):  
\[
[ q \in W; \ br'\text{-inner-invar} \ {r \in \delta}. q \in \text{ set } (r.hs r) } q (Q,W - \{ q \},rcm) } \{ } \Sigma' ]
\]
\[
\Rightarrow ((Q,W,rcm),\Sigma') \in br'\text{-step } \delta
\]

\langle proof \rangle

lemma \( br'\text{-inner-invar-step} \):
\[
[ q \in W; \ br'\text{-inner-invar} \ {r \in \delta}. q \in \text{ set } (r.hs r) } q (Q,W - \{ q \},rcm) } \text{ it } \Sigma';
\]
\[
r \in \text{ it}; \ i \in r \in \delta. q \in \text{ set } (r.hs r)
\]
\[
\Rightarrow br'\text{-inner-invar} \ {r \in \delta}. q \in \text{ set } (r.hs r) } q (Q,W - \{ q \},rcm)
\]
\[
(\text{it-}\{ r \}) (br'\text{-inner-step } r \Sigma')
\]

\langle proof \rangle

lemma \( br'\text{-inner-invar-initial} \):
\[
[ q \in W ] \Rightarrow br'\text{-inner-invar} \ {r \in \delta}. q \in \text{ set } (r.hs r) } q (Q,W - \{ q \},rcm)
\[
\quad \{ r \in \delta. q \in \text{ set } (r.hs r) } (Q,W - \{ q \},rcm)
\]

\langle proof \rangle

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

assumes \( \text{iterable-set} \): set\text{-iteratei } \alpha \text{ \text{invar} iteratei}

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

assumes \( \text{invar-step} \):
\[
!!it \ r. \Sigma. \ [ r \in \text{ it}; \ i \in r \in \delta. q \in \text{ set } (r.hs r) ]; \ \text{cinvar } it \Sigma ]
\]
\[
\Rightarrow \text{cinvar } (\text{it-}\{ r \}) (cstep r \Sigma)
\]

assumes \( \text{step-desc} \):
\[
!!it \ r. \Sigma. \ [ r \in \text{ it}; \ i \in \delta. q \in \text{ set } (r.hs r) ]; \ \text{cinvar } it \Sigma ]
\]
\[
\Rightarrow \alpha (cstep r \Sigma) = \text{br'\text{-inner-step } r } (\alpha \Sigma)
\]

assumes \( \text{it-set-desc} \): \text{invar } it\text{-set } \alpha it\text{-set } \{ r \in \delta. q \in \text{ set } (r.hs r) \}

assumes \( QIW[simp] \): \( q \in W \)
\textbf{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:

\textbf{definition} \textit{witness-prop} \( \delta \) \( m \) \( \equiv \forall q \ t. \ m q = \text{Some t} \rightarrow \text{accs} \ \delta \ t \ q \)

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

\textbf{definition} \textit{construct-witness} \( :: \ (\text{Q} \rightarrow \text{L tree}) \Rightarrow (\text{Q} \times \text{L} \times \text{ta-rule} \rightarrow \text{L tree}) \)

\textbf{where}

\textit{construct-witness} \( Q \ r \) \( \equiv \text{NODE} \ (\text{rhol} \ r) \ (\text{List.map} \ \lambda q. \ (\text{the} \ (Q \ q))) \ (\text{rhsq} \ r) \)

\textbf{lemma} \textit{witness-propD}: \[ \text{witness-prop} \ \delta \ Q; \ r \in \delta; \ \text{set} \ \text{rhsq} \ r \subseteq \text{dom} \ Q \] \( \implies \text{accs} \ \delta \ \text{construct-witness} \ Q \ r \ (\text{lhs} \ r) \)

\textbf{proof}

\textbf{lemma} \textit{construct-witness-correct}:

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

\textbf{proof}

\textbf{lemma} \textit{construct-witness-eq}:

\[ Q \ |\ \text{set} \ (\text{rhsq} \ r) = Q' \ |\ \text{set} \ (\text{rhsq} \ r) \] \( \implies \text{construct-witness} \ Q \ r = \text{construct-witness} \ Q' \ r \)

\textbf{proof}

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

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

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

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

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

\textbf{where} \textit{brw-invar-add} \( \delta \) \( \equiv \{(Q,W,rcm) . \ \text{witness-prop} \ \delta \ Q\} \)

\textbf{definition} \textit{brw-invar} \( \delta \) \( = \text{brw-invar-add} \ \delta \ \cap \{s. \ \text{brw-\alpha} \ s \in \text{br'-invar} \ \delta\} \)
inductive-set \texttt{brw-step}:: (\texttt{\textquotesingle Q,\textquotesingle L}) \texttt{ta-rule set} \Rightarrow ((\texttt{\textquotesingle Q,\textquotesingle L}) \texttt{brw-state} \times (\texttt{\textquotesingle Q,\textquotesingle L}) \texttt{brw-state}) \texttt{set} for \delta where

\[
\begin{align*}
q & \in W; \\
d_{sqr} = \{ r \in \delta. q \in \text{set} (\text{rhsq } r) \land \text{the} (\text{rcm } r) \leq 1 \}; \\
\text{dom } Q' = \text{dom } Q \cup \text{lhs}\_d_{sqr}; \\
\forall q t. \ (Q' q = \text{Some } t \iff Q q = \text{Some } t) \\
W' = (W - \{q\}) \cup (\text{lhs}\_d_{sqr} - \text{dom } Q); \\
\forall r \in \delta \Rightarrow \text{rcm } r' = (\text{if } q \in \text{set} (\text{rhsq } r) \text{ then} \\
\text{Some } (\text{the} (\text{rcm } r) - 1) \\
\text{else } \text{rcm } r) \\
\Rightarrow ((Q, W, \text{rcm}), (Q', W', \text{rcm }')) \in \texttt{brw-step } \delta
\end{align*}
\]

\texttt{definition \texttt{\textquotesingle Q set} \Rightarrow (\texttt{\textquotesingle Q,\textquotesingle L}) \texttt{brw-state set} where \texttt{brw-cond } Qi = \{(Q, W, \text{rcm}). W \neq \{\} \land (Qi \cap \text{dom } Q = \{\})\}

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

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

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

\[
\begin{align*}
\forall r \in \delta \Rightarrow \text{rcm } r = \text{Some } (\text{card } (\text{set } (\text{rhsq } r))); Q \in \texttt{brw-iq } \delta \\
\Rightarrow (Q, \texttt{brw-initial } \delta, \text{rcm}) \in \texttt{brw-initial } \delta
\end{align*}
\]

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

\texttt{lemma \texttt{brw-cond-abs}: \Sigma \in \texttt{brw-cond } Qi \iff (\texttt{brw-\alpha } \Sigma) \in \texttt{bre}'-\texttt{cond } Qi}

\texttt{lemma \texttt{brw-initial-abs}: \Sigma \in \texttt{brw-initial } \delta \Rightarrow \texttt{brw-\alpha } \Sigma \in \texttt{br}'-\texttt{initial } \delta}

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

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

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

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

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

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

theorem \texttt{brw-invar-final}:  
\(\forall \text{sc. } \text{sc} \in \text{wa-invar} (\text{brw-algo Qi} \delta) \land \text{sc} \notin \text{wa-cond} (\text{brw-algo Qi} \delta)\)  
\(\implies (\text{Qi} \land \text{dom} (\text{fst} \text{sc}) = \{\}) = (\text{Qi} \land \text{b-accessible} \delta = \{\})\)  
\(\land (\text{witness-prop} \delta (\text{fst} \text{sc}))\)  
\(\langle \text{proof} \rangle\)

Implementing a Step  
inductive-set \texttt{brw-inner-step}  
:: \((Q', L')\) ta-rule \(\Rightarrow\) ((\(Q', L')\) \text{brw-state} \(\times\) ((\(Q', L')\) \text{brw-state})) \text{set}\)  
for \(r\) where  
\[ \begin{array}{l} \text{c = the (rcm r); } \Sigma = (Q, W, rcm); \Sigma'=(Q', W', rcm'); \\
\text{if } c \leq 1 \land (\text{lhs} r) \notin \text{dom} Q \text{ then} \\
\quad Q' = Q(\text{lhs} r \mapsto \text{construct-witness} Q r) \\
\text{else } Q' = Q; \\
\text{if } c \leq 1 \land (\text{lhs} r) \notin \text{dom} Q \text{ then} \\
\quad W' = \text{insert} (\text{lhs} r) W \\
\text{else } W' = W; \\
\text{rcm'} = \text{rcm} (\ r \mapsto (c-(1::\text{nat}))) \end{array} \]  
\(\Rightarrow (\Sigma, \Sigma') \in \text{brw-inner-step} \ r\)

definition \texttt{brw-inner-invar}  
:: ((\(Q', L')\) ta-rule set \(\Rightarrow\) \(Q\) \(\Rightarrow\) ((\(Q', L')\) \text{brw-state} \(\Rightarrow\) ((\(Q', L')\) \text{brw-state} \(\Rightarrow\) bool) \text{where})  
\(\text{brw-inner-invar rules q} \equiv \lambda (Q, W, rcm) \text{ it} (Q', W', rcm').\)
lemma brw-inner-step-abs:

\((\Sigma,\Sigma') \in \text{brw-inner-step} \Rightarrow \text{brw-inner-step-abs} \ (\Sigma) = \text{brw-inner-step-abs} \ (\Sigma')\)

(\text{proof})

lemma brw-inner-invar-imp-final:

\([q \in W; \text{brw-inner-invar} \ \{r \in \delta. q \in \text{set} (\text{rhsq} r)\} \ q \ (Q,W-\{q\},rcm) \ {} \ {} \Sigma'] \]

\(\Rightarrow \ ((Q,W,rcm),\Sigma') \in \text{brw-step} \delta\)

(\text{proof})

lemma brw-inner-invar-step:

assumes INVI: \((Q,W,rcm) \in \text{brw-invar} \ \delta\)

assumes A: \(q \in W \ \ r \in \text{it} \ \ \ \text{it} \subseteq \{r \in \delta. q \in \text{set} (\text{rhsq} r)\}\)

assumes INVI: \(\text{brw-inner-invar} \ \{r \in \delta. q \in \text{set} (\text{rhsq} r)\} \ q \ (Q,W-\{q\},rcm) \ \text{it} \)

\Sigma h:: \(\Sigma\)

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

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

(\text{proof})

lemma brw-inner-invar-initial:

\([q \in W] \Rightarrow \text{brw-inner-invar} \ \{r \in \delta. q \in \text{set} (\text{rhsq} r)\} \ q \ (Q,W-\{q\},rcm) \ \{r \in \delta. q \in \text{set} (\text{rhsq} r)\} \ (Q,W-\{q\},rcm)\)

(\text{proof})

theorem brw-inner-step-proof:

\(\text{fixes as :: } \Sigma \Rightarrow (Q',L) \ \text{brw-state}\)

\(\text{fixes cstep :: } (Q',L) \ \text{ta-rule} \Rightarrow \Sigma \Rightarrow \Sigma\)

\(\text{fixes } \Sigma h:: \Sigma\)

\(\text{fixes cinvar :: } (Q',L) \ \text{ta-rule set} \Rightarrow \Sigma \Rightarrow \text{bool}\)

assumes set-iterate: \(\text{set-iterate} \alpha \ \text{invar iteratei}\)

assumes invar-start: \((\alpha \Sigma) \in \text{brw-invar} \ \delta\)

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

assumes invar-step:

\(!!it \ r. \Sigma. \ [r \in \text{it}; \ \text{it} \subseteq \{r \in \delta. q \in \text{set} (\text{rhsq} r)\}; \ \text{cinvar} \ \text{it} \ \Sigma] \]

\(\Rightarrow \ \text{cinvar} \ \text{it}-\{r\} \ \text{cstep} \ r \ \Sigma\)

assumes step-desc:

\(!!it \ r. \Sigma. \ [r \in \text{it}; \ \text{it} \subseteq \{r \in \delta. q \in \text{set} (\text{rhsq} r)\}; \ \text{cinvar} \ \text{it} \ \Sigma] \]

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

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

**assumes** \( QIW[\text{simp}]: q \in W \)

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

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

**shows** \( (\alpha \Sigma, \alpha \text{ (iterate it-set (} \lambda-. \text{ True} \) 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 == let \((Q,W,\delta) = S \text{ in } (Q, W) \)

**definition** \text{frp-invar-add} \( \delta1 \delta2 == \)
\\{(Q,W,\delta d). \delta d = \{ r. r \in \delta-prod \delta1 \delta2 \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 \ (ta-rules T1) \ (ta-rules T2) \}
\cap \{ s. \text{frp-\alpha} \ s \in \text{dfs-invar \ (ta-initial T1} \times \text{ta-initial T2) \}
\}

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

**inductive-set** frp-step

:: \((Q1,'L) \text { ta-rule set } \Rightarrow (Q2,'L) \text { ta-rule set} \)

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

for \( \delta1 \delta2 \) where

\[ W=\langle q1,q2 \rangle \# Wt; \]

\text{distinct} \ Wn;

\text{set} \ Wn = \text f\text{-succ (} \delta-prod \delta1 \delta2 \) \ (\{q1,q2\}) \ Q; \]

\( W'=Wn@Wt; \]

\( Q'=Q \cup \text f\text{-succ (} \delta-prod \delta1 \delta2 \) \ (\{q1,q2\}); \]

\( \delta d'=\delta d \cup \{ r \in \delta-prod \delta1 \delta2. \text{lhs } r = (q1,q2) \} \)

\[ \Rightarrow ((Q,W,\delta d),(Q',W',\delta d')) \in \text{frp-step } \delta1 \delta2 \]

**inductive-set** frp-initial :: \('Q1 \Rightarrow 'Q2 \Rightarrow ('Q1,'Q2,'L) \text { frp-state set} \)
for $Q10 \times Q20$ where
\[ \{ \text{distinct } W; \text{ set } W = Q10 \times Q20 \} \implies (Q10 \times Q20, W, \{\}) \in \text{frp-initial } Q10 \times Q20 \]

\textbf{definition} \textit{frp-cond} :: (\langle Q1, Q2, L \rangle) \textit{frp-state set} where
\[ \textit{frp-cond} == \{ (Q, W, \delta d). W \neq [\] } \]

\textbf{definition} \textit{frp-algo} $T1$ $T2$ == ()
\[ \textit{wa-cond} = \textit{frp-cond}, \]
\[ \textit{wa-step} = \textit{frp-step} (\textit{ta-rules} T1) (\textit{ta-rules} T2), \]
\[ \textit{wa-initial} = \textit{frp-initial} (\textit{ta-initial} T1) (\textit{ta-initial} T2), \]
\[ \textit{wa-invar} = \textit{frp-invar} T1 T2 \]

--- The algorithm refines the DFS-algorithm

\textbf{theorem} \textit{frp-pref-dfs}:
\[ \textit{wa-precise-refine} (\textit{frp-algo} T1 T2) \]
\[ (\textit{dfs-algo} (\textit{ta-initial} T1 \times \textit{ta-initial} T2) \]
\[ (f\textit{succ} (\delta \textit{prod} (\textit{ta-rules} T1) (\textit{ta-rules} T2)))) \]
\[ \textit{frp-\alpha} (proof) \]

\textbf{interpretation} \textit{frp-ref}:
\[ \textit{wa-precise-refine} (\textit{frp-algo} T1 T2) \]
\[ (\textit{dfs-algo} (\textit{ta-initial} T1 \times \textit{ta-initial} T2) \]
\[ (f\textit{succ} (\delta \textit{prod} (\textit{ta-rules} T1) (\textit{ta-rules} T2)))) \]
\[ \textit{frp-\alpha} (proof) \]

\textbf{theorem} \textit{frp-while-algo}:
\[ \textbf{assumes} TA: \textit{tree-automaton} T1 \]
\[ \textit{tree-automaton} T2 \]
\[ \textbf{shows} \textit{while-algo} (\textit{frp-algo} T1 T2) \]
\[ (proof) \]

\textbf{theorem} \textit{frp-inv-final}:
\[ \forall s. s \in \textit{wa-invar} (\textit{frp-algo} T1 T2) \land s \notin \textit{wa-cond} (\textit{frp-algo} T1 T2) \]
\[ \implies (\text{case } s \text{ of } (Q, W, \delta d) \Rightarrow \]
\[ (\begin{aligned} &| \textit{ta-initial} = \textit{ta-initial} T1 \times \textit{ta-initial} T2, \\
& \textit{ta-rules} = \delta d \\
& | = \textit{ta-fwd-reduce} (\textit{ta-prod} T1 T2) \end{aligned}) \]
\[ (proof) \]

end

\section{Executable Implementation of Tree Automata}

\textbf{theory} Ta-impl
\textbf{imports}
\[ \textit{Main} \]
\[ ../\text{Collections}/ICF/\text{CollectionsV1} \]
\[ Ta \textit{AbsAlgo} \]
\[ ~/src/HOL/Library/\text{Code-Target-Numeral} \]

40
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

\begin{verbatim}
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
\end{verbatim}

— Make wrapped states hashable

\begin{verbatim}
instantiation astate-wrapper :: (hashable, hashable) hashable
begin
  definition hashcode x == (case x of USW1 a ⇒ 2 * hashcode a | USW2 b ⇒ 2 * hashcode b + 1)
  definition def-hashmap-size = (λ- :: ('a,'b) astate-wraper) itself. def-hashmap-size TYPE('a) + def-hashmap-size TYPE('b))

instance ⟨proof⟩
end
\end{verbatim}

5.1.1 Ad-Hoc instantiations of generic Algorithms

\begin{verbatim}
⟨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⟩
⟨proof⟩
interpretation lh-set-xx!: g-set-xx-loc ls-ops hs-ops ⟨proof⟩
interpretation lll-iflt-cp: inj-image-filter-cp-loc ls-ops ls-ops ls-ops ⟨proof⟩
\end{verbatim}
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⟩
 interpretaton lh-set-xy!: g-set-xy-loc ls-ops hs-ops hs-ops ⟨proof⟩
interpretation hh-map-to-nat!: map-to-nat-loc hs-ops hm-ops ⟨proof⟩
interpretation lh-set-xy!: g-set-xy-loc ls-ops hs-ops ⟨proof⟩
interpretation hh-set-xx!: g-set-xx-loc hs-ops hs-ops ⟨proof⟩
interpretation hs-to-fifo!: set-to-list-loc hs-ops fifo-ops ⟨proof⟩

⟨ML⟩

5.2 Generating Indices of Rules

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

f index of rules by function symbol

s index of rules by lhs

sf index of rules

definition build-rule-index
:: ((‘q,’l) ta-rule ⇒ ‘i::hashable) ⇒ (‘q,’l) ta-rule ls
⇒ (‘i,(‘q,’l) ta-rule ls) hm

where build-rule-index == hll-idx.idx-build

definition build-rule-index-f δ == build-rule-index (λr. rhsl r) δ

definition build-rule-index-s δ == build-rule-index (λr. lhs r) δ

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

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

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

lemma build-rule-index-sf-correct[simp]:
assumes I[simp, intro!]: ls-invar δ
shows
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 \| hta-idx-f := Some (build-rule-index-f (hta-δ H)) \|
  Some - ⇒ H

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

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

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

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

proof

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

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

definition hta-has-idx-f H == hta-idx-f H \neq \text{None}
  — Check whether the f-index is present

definition hta-has-idx-s H == hta-idx-s H \neq \text{None}
  — Check whether the s-index is present

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

lemma hta-idx-f-pres
  [simp, intro!]: hta-has-idx-f (hta-ensure-idx-f H) and
  [simp, intro]: hta-has-idx-s H \implies hta-has-idx-f (hta-ensure-idx-f H) and
  [simp, intro]: hta-has-idx-sf H \implies hta-has-idx-f (hta-ensure-idx-f H)
(proof)

lemma hta-idx-s-pres
  [simp, intro!]: hta-has-idx-s (hta-ensure-idx-s H) and
  [simp, intro]: hta-has-idx-f H \implies hta-has-idx-s (hta-ensure-idx-s H) and
  [simp, intro]: hta-has-idx-sf H \implies hta-has-idx-s (hta-ensure-idx-s H)
(proof)

lemma hta-idx-sf-pres
  [simp, intro!]: hta-has-idx-sf (hta-ensure-idx-sf H) and
  [simp, intro]: hta-has-idx-f H \implies hta-has-idx-sf (hta-ensure-idx-sf H) and
  [simp, intro]: hta-has-idx-s H \implies hta-has-idx-sf (hta-ensure-idx-sf H)
(proof)

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

  — Lookup rules by function symbol

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

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

definition hta-lookup-sf q f H == hll-idx.lookup (q, f) (the (hta-idx-sf H))
— This locale defines the invariants of a tree automaton

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

— The involved sets satisfy their invariants

assumes invar[simp, intro!]:
  hs-invar (hta-Qi H)
  ls-invar (hta-δ H)

— The indices are correct, if present

assumes index-correct:
  hta-idx-f H = Some idx-f
  ==> hll-idx.is-index rhsl (ls-α (hta-δ H)) idx-f
  hta-idx-s H = Some idx-s
  ==> hll-idx.is-index lhs (ls-α (hta-δ H)) idx-s
  hta-idx-sf H = Some idx-sf
  ==> hll-idx.is-index (λr. (lhs r, rhsl r)) (ls-α (hta-δ H)) idx-sf

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

abbreviation δ == hta-δ H
abbreviation Qi == hta-Qi H

— The lookup-xxx operations are correct

lemma hta-lookup-f-correct:
  hta-has-idx-f H ==> ls-α (hta-lookup-f f H) = {r∈ls-α δ . rhsl r = f}
  hta-has-idx-f H ==> ls-invar (hta-lookup-f f H)
⟨proof⟩

lemma hta-lookup-s-correct:
  hta-has-idx-s H ==> ls-α (hta-lookup-s q H) = {r∈ls-α δ . lhs r = q}
  hta-has-idx-s H ==> ls-invar (hta-lookup-s q H)
⟨proof⟩

lemma hta-lookup-sf-correct:
  hta-has-idx-sf H
  ==> ls-α (hta-lookup-sf q f H) = {r∈ls-α δ . lhs r = q ∧ rhsl r = f}
  hta-has-idx-sf H ==> ls-invar (hta-lookup-sf q f H)
⟨proof⟩

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

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

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

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

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

end

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

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

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

5.4 Algorithm for the Word Problem

lemma r-match-by-laz: r-match L l = list-all-zip (λQ q ∈ Q) 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 H1[simp, intro!]: hta-has-idx-f H
  shows ls-invar (faccs' H t) (is ?T1)
    list-all ls-invar (List.map (faccs' H) ts) (is ?T2)
  ⟨proof⟩

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

lemma faccs'-correct:
  assumes H1[simp, intro!]: hta-has-idx-f H
  shows
    ls-α (faccs' H t) = faccs (ls-α (hta-δ H)) t (is ?T1)
    List.map ls-α (List.map (faccs' H) ts)
    = List.map (faccs (ls-α (hta-δ H))) ts (is ?T2)
  ⟨proof⟩

lemma hta-mem'-correct:
  hta-has-idx-f H ⇒ hta-mem' t H ←→ t ∈ ta-lang (hta-α H)
  ⟨proof⟩

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

end

5.5 Product Automaton and Intersection
5.5.1 Brute Force Product Automaton

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

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

lemma r-prod-inj:
[ rhsl r1 = rhsl r2; length (rhsq r1) = length (rhsq r2);
  rhsl r1' = rhsl r2'; length (rhsq r1') = length (rhsq r2');
  r-prod r1 r2 = r-prod r1' r2' ] ⇒ r1=r1' ∧ r2=r2'
lemma δ-prod-h-correct:
assumes INV[simp]: ls-invar δ1 ls-invar δ2
shows
ls-α (δ-prod-h δ1 δ2) = δ-prod (ls-α δ1) (ls-α δ2)
ls-invar (δ-prod-h δ1 δ2)
(proof)

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

lemma hta-prodWR-correct-aux:
assumes A: hashedTa H1 hashedTa H2
shows
hta-α (hta-prodWR H1 H2) = ta-prod (hta-α H1) (hta-α H2) (is ?T1)
hashedTa (hta-prodWR H1 H2) (is ?T2)
(proof)

lemma hta-prodWR-correct:
assumes TA: hashedTa H1 hashedTa H2
shows
ta-lang (hta-α (hta-prodWR H1 H2))
= ta-lang (hta-α H1) ∩ ta-lang (hta-α H2)
hashedTa (hta-prodWR H1 H2)
(proof)

5.5.2 Product Automaton with Forward-Reduction

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

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

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

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

— Adds all successor states to the set of discovered states and to the worklist
fun pa-upd-rule
:: (′q1×′q2) hs ⇒ (′q1×′q2) list
\[ (('q1::hashable) \times ('q2::hashable)) \text{ list} \]
\[ (('q1 \times 'q2) \text{ hs } \times ('q1 \times 'q2) \text{ list}) \]

**where**

\[
\text{pa-upd-rule } Q \ W \ [] = (Q, W) |
\]

\[
\text{pa-upd-rule } Q \ W \ (qp\#qs) = (\]
\[ \text{if } \neg \text{hs-memb qp Q then}
\]
\[ \text{pa-upd-rule } (\text{hs-ins qp Q}) (qp\#W) \text{ qs}
\]
\[ \text{else } \text{pa-upd-rule } Q \ W \text{ qs}
\]
\[
\)

**definition** \text{pa-step} :: ('q1::hashable,'l::hashable) \text{ hashedTa} \Rightarrow ('q2::hashable,'l) \text{ hashedTa} \Rightarrow ('q1,'q2,'l) \text{ pa-state} \Rightarrow ('q1,'q2,'l) \text{ pa-state} \]

**where** \text{pa-step } H1 H2 \text{ S} == let
\[ (Q,W,\delta d)=S;
\]
\[ (q1,q2)=\text{hd } W
\]
\[
\text{in \text{ls-iteratei} (hta-lookup-s q1 H1) (\lambda-. \text{True}) (\lambda r1 \text{ res.})}
\]
\[
\text{ls-iteratei} (hta-lookup-sf q2 (rhsq r1) H2) (\lambda-. \text{True}) (\lambda r2 \text{ res.})
\]
\[ \text{if } (\text{length (rhsq r1) } = \text{length (rhsq r2)}) \text{ then}
\]
\[ \text{let}
\]
\[ rp=r-prod r1 \ r2;
\]
\[ (Q,W,\delta d) = \text{res};
\]
\[ (Q',W') = \text{pa-upd-rule } Q \ W \text{ (rhsq rp)}
\]
\[
\text{in}
\]
\[ (Q',W',\text{ls-ins-dj } rp \ \delta d)
\]
\[ \text{else}
\]
\[ \text{res}
\]
\[ ) \quad (Q,l W,\delta d)
\]

**definition** \text{pa-initial} :: ('q1::hashable,'l::hashable) \text{ hashedTa} \Rightarrow ('q2::hashable,'l) \text{ hashedTa} \Rightarrow ('q1,'q2,'l) \text{ pa-state} \]

**where** \text{pa-initial } H1 H2 ==
\[ \text{let } Qip = \text{hhb-cart.cart} (hta-Qi H1) (hta-Qi H2) \text{ in (}
\]
\[ Qip,
\]
\[ \text{hs-to-list } Qip,
\]
\[ \text{ls-empty }()
\]
\[ )
\]

**definition** \text{pa-invar-add} ::
\[ ('q1::hashable,'q2::hashable,'l::hashable) \text{ pa-state set} \]

**where** \text{pa-invar-add} == \{ (Q,W,\delta d). \text{hs-invar } Q \land \text{ls-invar } \delta d \}
definition \textit{pa-invar} H1 H2 ==
\[
\text{pa-invar-add} \cap \{ s. (\text{pa-}\alpha s) \in \text{frp-invar} (hta-\alpha H1)(hta-\alpha H2) \}
\]

definition \textit{pa-det-algo} H1 H2 ==
\[
\emptyset \text{ wwa-cond}=\text{pa-cond}, \quad wwa-step = \text{pa-step} H1 H2, \quad wwa-initial = \text{pa-initial} H1 H2, \quad wwa-invar = \text{pa-invar} H1 H2 \]

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

lemmas \textit{pa-det-while-algo} = \textit{det-while-algo-intro}[OF \textit{pa-while-algo}]

— Transferred correctness lemma

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

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’\( H1 \ H2 \) ==

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

**definition** hta-prod \( H1 \ H2 \) ==

\[hta-prod’ (hta-ensure-idx-s H1) (hta-ensure-idx-sf H2)\]

**lemma** hta-prod’-correct-aux:

**assumes** TA: htedTa H1 htedTa H2

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

**shows** hta-\( \alpha \) (hta-prod’ H1 H2)

\[= \text{ta-fwd-reduce} (\text{ta-prod} (hta-\( \alpha \) H1) (hta-\( \alpha \) H2)) (\text{is } ?T1)\]

\[\text{htedTa} (hta-prod’ H1 H2) (\text{is } ?T2)\]

(proof)

**theorem** hta-prod’-correct:

**assumes** TA: htedTa H1 htedTa H2

**assumes** HI: hta-has-idx-s H1 hta-has-idx-sf H2

**shows**

\[\text{ta-lang} (hta-\( \alpha \) (hta-prod’ H1 H2))\]

\[= \text{ta-lang} (hta-\( \alpha \) H1) \cap \text{ta-lang} (hta-\( \alpha \) H2)\]

\[\text{htedTa} (hta-prod’ H1 H2)\]

(proof)

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

**assumes** TA[simp]: htedTa H1 htedTa H2

**shows**

\[hta-\( \alpha \) (hta-prod H1 H2) = \text{ta-fwd-reduce} (\text{ta-prod} (hta-\( \alpha \) H1) (hta-\( \alpha \) H2))\]

\[\text{htedTa} (hta-prod H1 H2)\]

(proof)

**theorem** hta-prod-correct:

**assumes** TA: htedTa H1 htedTa H2

**shows**

\[\text{ta-lang} (hta-\( \alpha \) (hta-prod H1 H2))\]

\[= \text{ta-lang} (hta-\( \alpha \) H1) \cap \text{ta-lang} (hta-\( \alpha \) H2)\]

\[\text{htedTa} (hta-prod H1 H2)\]

(proof)
5.6 Remap States

— Mapping the states of an automaton

**Definition hta-remap**

\[
\text{hta-remap} :: (\text{'q::hashable} \Rightarrow (\text{'qn::hashable} \Rightarrow (\text{'ql::hashable}) \text{hashedTa})} \\
\text{where hta-remap f H ==} \\
\text{init-hta (hh-set-xy.g-image f (hta-Qi H))} \\
(\text{ll-set-xy.g-image (remap-rule f) (hta-δ H))}
\]

**Lemma (in hashedTa) hta-remap-correct**: shows hta-α (hta-remap f H) = ta-remap f (hta-α H)

\langle proof \rangle

5.6.1 Reindex Automaton

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

— Set of states of a rule

**Function rule-states-l where**

\[
\text{rule-states-l (q → f qs) = ls-ins q (ls.from-list qs)}
\]

**Lemma rule-states-l-correct\[\text{simp\}:**

\[
\text{ls-α (rule-states-l r) = rule-states r} \\
\text{ls-invar (rule-states-l r)}
\]

\langle proof \rangle

**Definition hta-δ-states H**

\[
\text{hta-δ-states H == (llh-set-xyy.g-Union-image id (ll-set-xy.g-image-filter (λr. Some (rule-states-l r))) (hta-δ H)))}
\]

**Definition hta-states H ==**

\[
\text{hs-union (hta-Qi H) (hta-δ-states H)}
\]

**Lemma (in hashedTa) hta-δ-states-correct**: shows hta-α (hta-δ-states H) = δ-states (ta-rules (hta-α H))

\langle proof \rangle

**Lemma (in hashedTa) hta-states-correct**: shows hta-α (hta-states H) = ta-rstates (hta-α H)

\langle proof \rangle

**Definition reindex-map H ==**

\[
\lambda q. \text{the (hm-lookup q (hh-map-to-nat.map-to-nat (hta-states H)))}
\]

52
definition hta-reindex
:: (′Q::hashable,′L::hashable) hashedTa ⇒ (nat,′L) hashedTa where
hta-reindex H == hta-remap (reindex-map H) H

declare hta-reindex-def [code del]

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

lemma [code]: hta-reindex H = (let mp = (hh-map-to-nat.map-to-nat (hta-states H)) in
hta-remap (λq. the ((hm-lookup q mp)) H)

⟨proof⟩

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

theorem (in hashedTa) hta-reindex-correct:
ta-lang (hta-α (hta-reindex H)) = ta-lang (hta-α H)
hashedTa (hta-reindex H)
⟨proof⟩

5.7 Union

Computes the union of two automata

definition hta-union
:: (′q1::hashable,′l::hashable) hashedTa ⇒ (′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)))
(lq-union-dj (ll-set-xy,g-image (remap-rule USW1) (hta-δ H1))
(ll-set-xy,g-image (remap-rule USW2) (hta-δ H2)))

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

theorem hta-union-correct:
assumes TA: hashedTa H1 hashedTa H2
shows
ta-lang (hta-α (hta-union H1 H2))
== ta-lang (hta-α H1) ∪ ta-lang (hta-α H2) (is ?T1)
hashedTa (hta-union H1 H2) (is ?T2)
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

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

shows (hta-α (hta-empty ())) = ta-empty

hashedTa (hta-empty ())

⟨proof⟩

— Add a rule to the automaton

**Definition** hta-add-rule :: ('q,'l) ta-rule ⇒ ('q::hashable,'l::hashable) hashedTa

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

shows hta-α (hta-add-rule r H)

= (\) ta-initial = insert r (ta-initial (hta-α H)),

\ta-rules = ta-rules (hta-α H)

\)

hashedTa (hta-add-rule r H)

⟨proof⟩

— 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

54
(λr. if hs-memb (lhs r) Q ∧ list-all (λq. hs-memb q Q) (rhsq r) then Some r else None) (hta-δ H))

theorem (in hashedTa) hta-reduce-correct:
  assumes INV[simp]: hs-invar Q
  shows hta-α (hta-reduce H Q) = ta-reduce (hta-α H) (hs-α Q) (is ?T1)
  hashedTa (hta-reduce H Q) (is ?T2)
(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,'l) ta-rule ls hm
where
build-rqrm δ ==
ls-iterate δ (λ-. True)
(λr res.
  foldl (λres q. rqrm-add q r res) res (rhsq r))
) (hm-empty ())

— Whether the index satisfies the map and set invariants
definition rqrm-invar rqrm ==
hm-invar rqrm ∧ (∀q. ls-invar (rqrm-lookup rqrm q))
— Whether the index really maps a state to the set of rules with this state on their rhs
definition rqrm-prop δ rqrm ==
∀q. ls-α (rqrm-lookup rqrm q) = {r∈δ. q∈set (rhsq r)}
lemma rqrm-α-lookup-update[simp]:
  rqrm-invar rqrm \implies
  ls-α (rqrm-lookup (rqrm-add q r rqrm) q′)
  = ( if q=q′ then
       insert r (ls-α (rqrm-lookup rqrm q′))
     else
       ls-α (rqrm-lookup rqrm q′)
  )

⟨proof⟩

lemma rqrm-propD:
  rqrm-prop δ rqrm \implies
  ls-α (rqrm-lookup rqrm q) = \{ r ∈ δ. q ∈ set (rhsq r) \}

⟨proof⟩

lemma build-rqrm-correct:
  fixes δ
  assumes [simp]: ls-invar δ
  shows rqrm-invar (build-rqrm δ) (is ?T1)
    and rqrm-prop (ls-α δ) (build-rqrm δ) (is ?T2)

⟨proof⟩

type-synonym ('Q,'L) brc-state
  = 'Q hs × 'Q list × ('Q,'L ta-rule, nat) hm

— Abstraction to α'-level:
definition brc-α :: ('Q::hashable,'L::hashable) brc-state ⇒ ('Q,'L) br'-state
  where brc-α \equiv 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 \equiv \{ (Q,W,rcm).
    hs-invar Q ∧
    distinct W ∧
    hm-invar rcm
    (\* ∧ set W ⊆ hs-α Q \*) \}
definition brc-invar δ \equiv brc-invar-add ∩ \{ s. brc-α s ∈ br'-invar δ \}
definition brc-cond :: ('q::hashable,'l::hashable) brc-state ⇒ bool
  where brc-cond \equiv λ(Q,W,rcm). W \not\equiv []
definition brc-inner-step :: ('q,'l) ta-rule ⇒ ('q::hashable,'l::hashable) brc-state
  \⇒ ('q,'l) brc-state
  where
  brc-inner-step r \equiv λ(Q,W,rcm).
  let c=the (hm-lookup r rcm);
\[
rcm' = hm\text{-update } r (c \cdot (1::\text{nat})) rcm;
Q' = (\text{if } c \leq 1 \text{ then } hs\text{-ins } (lhs \ r) \ Q \text{ else } Q);
W' = (\text{if } c \leq 1 \land \neg hs\text{-memb } (lhs \ r) \ Q \text{ then } lhs \ r \# W \text{ else } W) \text{ in } (Q', W', rcm')
\]

**definition brc-step**
\[
\text{:: } (\text{'}q, (\text{'}q, \text{'}l) \text{-ta-rule } ls) \text{-rule } hm
\Rightarrow (\text{'}q::\text{hashable}, \text{'}l::\text{hashable}) \text{ brc-state}
\Rightarrow (\text{'}q, \text{'}l) \text{ brc-state}
\]

**where**
\[
\text{brc-step rqrm} = \lambda (Q, W, rcm). 
\text{ls-iteratei } (rqrm\text{-lookup } rqrm (hd W)) \ (\lambda -. \text{True}) \text{ brc-inner-step } 
(Q, tl W, rcm)
\]

---

**definition brc-iq** :: (\text{'}q, \text{'}l) \text{-ta-rule } ls \Rightarrow \text{'}q::\text{hashable} \text{ hs}

**where**
\[
\text{brc-iq } \delta = \text{lh-set-xy, g-image-filter } (\lambda r. 
\text{if } rhsq r = [] \text{ then } \text{Some } (lhs r) \text{ else } \text{None}) \ \delta
\]

**definition brc-rcm-init** :: (\text{'}q::\text{hashable}, \text{'}l::\text{hashable}) \text{-ta-rule } ls \Rightarrow ((\text{'}q, \text{'}l) \text{-ta-rule, nat}) \text{-rule } hm

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

**definition brc-initial** :: (\text{'}q::\text{hashable}, \text{'}l::\text{hashable}) \text{-ta-rule } ls \Rightarrow (\text{'}q, \text{'}l) \text{ brc-state}

**where**
\[
\text{brc-initial } \delta = 
\text{let iq= brc-iq } \delta \text{ in}
(iq, hs\text{-to-list } (iq), \text{ brc-rcm-init } \delta)
\]

**definition brc-det-algo rqrm \delta = ()
\[
dwa-cond = \text{brc-cond},
dwa-step = \text{brc-step rqrm},
dwa-initial = \text{brc-initial } \delta,
dwa-invar = \text{brc-invar } (ls\cdot \alpha \ \delta)
\]

---

Additional facts needed from the abstract level

**lemma brc-ine-imp-WssQ**: \( \text{brc-\alpha } (Q, W, rcm) \in br'\text{-invar } \delta \Rightarrow \text{set } W \subseteq hs\cdot \alpha \ Q \)

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

**lemma brc-iq-correct**:

**assumes** [simp]: \( \text{ls\text{-invar } } \delta \)

**shows** \( \text{hs\text{-invar } } (\text{brc-iq } \delta) \)
\[
\text{hs}\cdot \alpha \ (\text{brc-iq } \delta) = \text{br}\cdot \text{-iq } (ls\cdot \alpha \ \delta)
\]

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

lemma \textit{brc-inner-step-br'}-desc:
\begin{align*}
\text{(Q,W,rcm)} \in \text{brc-invar } \delta \quad \implies \quad & \text{brc-} \alpha (\text{brc-inner-step } \ r \ (Q,W,rcm)) = ( \\
& \quad \text{if the } (\text{hm-} \alpha \ \text{rcm } r) \leq 1 \text{ then} \\
& \quad \quad \text{insert } (\text{lhs } r) \ (\text{hs-} \alpha \ Q) \\
& \quad \text{else } \text{hs-} \alpha \ Q, \\
& \quad \text{if the } (\text{hm-} \alpha \ \text{rcm } r) \leq 1 \land (\text{lhs } r) \notin \text{hs-} \alpha \ Q \text{ then} \\
& \quad \quad \text{insert } (\text{lhs } r) \ (\text{set } W) \\
& \quad \text{else } (\text{set } W), \\
& \quad ((\text{hm-} \alpha \ \text{rcm}) (r \mapsto \text{the } (\text{hm-} \alpha \ \text{rcm } r) - 1))
\}
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-step-invar}:
\begin{align*}
\text{assumes } & \text{RQRM: } \text{rqrm-invar } \text{rqrm} \\
\text{shows } & [ \Sigma \in \text{brc-invar-add}; \text{brc-} \alpha \ \Sigma \in \text{br'}-\text{invar } \delta; \text{brc-cond } \Sigma ] \\
& \implies (\text{brc-step } \text{rqrm } \Sigma) \in \text{brc-invar-add} \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-step-abs}:
\begin{align*}
\text{assumes } & \text{RQRM: } \text{rqrm-invar } \text{rqrm} \quad \text{rqrm-prop } \delta \ \text{rqrm} \\
\text{assumes } & A \colon \Sigma \in \text{brc-invar } \delta \quad \text{brc-cond } \Sigma \\
\text{shows } & (\text{brc-} \alpha \ \Sigma, \text{brc-} \alpha (\text{brc-step } \text{rqrm } \Sigma)) \in \text{br'}-\text{step } \delta \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-initial-invar}:
\begin{align*}
\text{ls-invar } \delta \quad \implies \quad & (\text{brc-initial } \delta) \in \text{brc-invar-add} \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-cond-abs}:
\begin{align*}
\text{brc-cond } \Sigma \quad \iff \quad & (\text{brc-} \alpha \ \Sigma) \in \text{br'}-\text{cond} \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-initial-abs}:
\begin{align*}
\text{ls-invar } \delta \quad \implies \quad & \text{brc-} \alpha (\text{brc-initial } \delta) \in \text{br'}-\text{initial } (\text{ls-} \alpha \ \delta) \\
\langle \text{proof} \rangle
\end{align*}

lemma \textit{brc-pref-br'}:
\begin{align*}
\text{assumes } & \text{RQRM[simp]: } \text{rqrm-invar } \text{rqrm} \quad \text{rqrm-prop } (\text{ls-} \alpha \ \delta) \ \text{rqrm} \\
\text{assumes } & \text{INV[simp]: } \text{ls-invar } \delta \\
\text{shows } & \text{wa-precise-refine } (\text{det-wa-wa } (\text{brc-det-algo } \text{rqrm } \delta)) \\
& \quad (\text{br'}-\text{algo } (\text{ls-} \alpha \ \delta))
\end{align*}

58
\( \text{brc-\(\alpha\)} \)

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

**lemma** \( \text{brc-while-algo} \):

**assumes** \( RQRM[\text{simp}]: \text{rqrm-invar \ rqrm \ \text{rqrm-prop (ls-\(\alpha\) \(\delta\)) \ rqrm} \)

**assumes** \( INV[\text{simp}]: \text{ls-invar} \(\delta\) \)

**shows** \( \text{while-algo (det-\text{wa-wa} (\text{brc-det-algo \ rqrm \(\delta\)}) \text{)} } \)

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

**lemmas** \( \text{brc-det-while-algo} = \)

\( \text{det-while-algo-intro}[\text{OF \ brc-while-algo]} \)

**lemma** \( \text{fst-brc-\(\alpha\): \text{fst (brc-\(\alpha\) } s) = \text{hs-\(\alpha\) (fst s)} \text{)} \)

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

**lemmas** \( \text{brc-invar-final} = \)

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

**definition** \( \text{hta-bwd-reduce } H \)

\( \text{let \rqrm = build-rqrm (hta-\(\delta\) } H \text{ in} \text{)} \)

\( \text{hta-reduce } H \)

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

**theorem** (in \( \text{hashedTa} \)) \( \text{hta-bwd-reduce-correct} \):

**shows** \( \text{hta-\(\alpha\) } (\text{hta-bwd-reduce } H) \)

\( = \text{ta-reduce } (\text{hta-\(\alpha\) } H) (\text{b-accessible (ls-\(\alpha\) (hta-\(\delta\) } H))} \text{ (is } ?T1) \text{)} \)

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

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

### 5.9.1 Emptiness Check with Witness Computation

**definition** \( \text{brec-construct-witness} \)

\( :: ('q::hashable, 'l::hashable tree) \text{ hm } \Rightarrow ('q, 'l) \text{ ta-rule } \Rightarrow 'l \text{ tree} \)

**where** \( \text{brec-construct-witness } Qm \text{ r } = \text{NODE (rhsr r) (List.map (\lambda q. } \text{the (hm-lookup } q \text{ Qm)}) \text{ (rhsq r)}} \)

**lemma** \( \text{brec-construct-witness-correct} \):

\( [\text{hm-invar } Qm] \Rightarrow \text{brec-construct-witness } Qm \text{ r } = \text{construct-witness (hm-\(\alpha\) Qm) } r \text{)} \)

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

**type-synonym** \( ('Q,'L) \text{ brec-state} \)

\( = (('Q,'L \text{ tree}) \text{ hm} \times 'Q \text{ fifo} \times (('Q,'L) \text{ ta-rule, nat) } hm \text{)} \)

59
× 'Q option')

— Abstractions

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

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

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

definition brec-igm δ ==
 ls-iteratei δ (λ-. True) (λr m. if rhsq r = [] then
    hm-update (lhs r) (NODE (rhsl r) []) m
  else m)
 (hm-empty ())

definition brec-initial :: 'q hs ⇒ ('q::hashable,'l::hashable) ta-rule ls
 ⇒ ('q,'l) brec-state
 where brec-initial Qi δ ==
 let iq=brc-ig δ in
 ( brec-igm δ,
   hs-to-fifo.g-set-to-listr iq,
definition brec-cond
:: ('q,'l) brec-state ⇒ bool
where brec-cond == \( \lambda (Q,W,rcm,qwit). \) \&\& fifo-isEmpty \( W \) \&\& quwit = None

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

definition brec-invar \( Qi \) \( δ \) == brec-invar-add \( Qi \) \&\& \{ \text{s. brec-α } s \in \text{brw-invar } δ \}

definition brec-invar-inner \( Qi \) \( δ \) ==
  brec-invar-add \( Qi \) \&\& \{ \text{s. set } (\text{fifo-α } W) \subseteq \text{dom } (\text{hm-α } Q) \}

lemma brec-invar-cons:
\[ \Sigma \in brec-invar \( Qi \) \( δ \) \implies \Sigma \in brec-invar-inner \( Qi \) \]
(proof)

lemma brec-brw-invar-cons:
\[ \text{brec-α } \Sigma \in \text{brw-invar } Qi \implies \text{set } (\text{fifo-α } (\text{fst } \Sigma)) \subseteq \text{dom } (\text{hm-α } (\text{fst } \Sigma)) \]
(proof)

definition brec-det-algo rqrn \( Qi \) \( δ \) ==
  \{ \text{dwa-cond=brec-cond,}
    \text{dwa-step=brec-step rqrn } Qi,\]
  \text{dwa-initial=brec-initial } Qi \( δ),\]
  \text{dwa-invar=brec-invar } (\text{hs-α } Qi) (\text{ls-α } δ)\}

lemma brec-igm-correct:\nassumes \text{INV[simp]: } \text{ls-invar } δ\nsows
\[ \text{dom } (\text{hm-α } (\text{brec-igm } δ)) = \{ \text{lhs } r \mid r. \ r \in \text{ls-α } δ \land \text{rhsq } r = [] \} \]
(is \?T1)
\[ \text{witness-prop } (\text{ls-α } δ) (\text{hm-α } (\text{brec-igm } δ)) \]
(is \?T2)
\[ \text{hm-invar } (\text{brec-igm } δ) \]
(is \?T3)
(proof)

lemma brec-igm-correct:\nassumes \text{INV[simp]: } \text{ls-invar } δ
shows \( \text{hm-} \alpha (\text{brec-iqm} \ \delta) \in \text{brw-iq} (\text{ls-} \alpha \ \delta) \)

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

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

\[
[ \Sigma \in \text{brec-invar-inner} (\text{hs-} \alpha \ \text{Qi}) ] \Rightarrow (\text{brec-} \alpha \ \Sigma, \text{brec-} \alpha (\text{brec-inner-step} \ \text{Qi} \ \Sigma)) \in \text{brw-inner-step} \ \Sigma
\]

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

**Lemma brec-step-invar:**

\[
\begin{align*}
\text{assumes } & R\text{QRM}: \text{rqrm-invar} \ \text{rqrm} \ \text{rqrm-prop} \ \delta \ \text{rqrm} \\
\text{assumes } & \text{INV}[\text{simp}]: \text{hs-invar} \ \text{Qi} \\
\text{shows } & [ \Sigma \in \text{brec-invar-add} (\text{hs-} \alpha \ \text{Qi}); \text{brec-} \alpha \ \Sigma \in \text{brw-invar} \ \delta; \text{brec-cond} \ \Sigma ] \Rightarrow (\text{brec-step} \ \text{rqrm} \ \text{Qi} \ \Sigma) \in \text{brec-invar-add} (\text{hs-} \alpha \ \text{Qi})
\end{align*}
\]

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

**Lemma brec-invar-initial:**

\[
[ [ \text{ls-invar} \ \delta; \text{hs-invar} \ \text{Qi} ] ] \Rightarrow (\text{brec-initial} \ \text{Qi} \ \delta) \in \text{brec-invar-add} (\text{hs-} \alpha \ \text{Qi})
\]

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

**Lemma brec-cond-abs:**

\[
[ [ \Sigma \in \text{brec-invar} \ \text{Qi} \ \delta ] ] \Rightarrow \text{brec-cond} \ \Sigma \leftrightarrow (\text{brec-} \alpha \ \Sigma) \in \text{brw-cond} \ \text{Qi}
\]

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

**Lemma brec-initial-abs:**

\[
[ [ \text{ls-invar} \ \delta; \text{hs-invar} \ \text{Qi} ] ] \Rightarrow (\text{brec-initial} \ \text{Qi} \ \delta) \in \text{brw-initial} (\text{ls-} \alpha \ \delta)
\]

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

**Lemma brec-prefix-brw:**

\[
\begin{align*}
\text{assumes } & R\text{QRM}[\text{simp}]: \text{rqrm-invar} \ \text{rqrm} \ \text{rqrm-prop} (\text{ls-} \alpha \ \delta) \ \text{rqrm} \\
\text{assumes } & \text{INV}[\text{simp}]: \text{ls-invar} \ \delta \ \text{hs-invar} \ \text{Qi} \\
\text{shows } & \text{wa-precise-refine} (\text{det-wa-wa (brec-det-algo} \ \text{rqrm} \ \text{Qi} \ \delta)) (\text{brw-algo} (\text{hs-} \alpha \ \text{Qi} (\text{ls-} \alpha \ \delta)) \text{brec-} \alpha
\end{align*}
\]

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

**Lemma brec-while-algo:**

\[
\begin{align*}
\text{assumes } & R\text{QRM}[\text{simp}]: \text{rqrm-invar} \ \text{rqrm} \ \text{rqrm-prop} (\text{ls-} \alpha \ \delta) \ \text{rqrm} \\
\text{assumes } & \text{INV}[\text{simp}]: \text{ls-invar} \ \delta \ \text{hs-invar} \ \text{Qi} \\
\text{shows } & \text{while-algo} (\text{det-wa-wa (brec-det-algo} \ \text{rqrm} \ \text{Qi} \ \delta)) (\text{brec-} \alpha
\end{align*}
\]

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

62
⟨proof⟩

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

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

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

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

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

5.10 Interface for Natural Number States and Symbols

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

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

type-synonym hta = (nat,nat) hashedTa

definition hta-mem :: - ⇒ hta ⇒ bool
  where hta-mem == hta-mem
definition hta-prod :: hta ⇒ hta ⇒ hta
  where hta-prod H1 H2 == hta-reindex (hta-prod H1 H2)
definition hta-prodWR :: hta ⇒ hta ⇒ hta
  where hta-prodWR H1 H2 == hta-reindex (hta-prodWR H1 H2)
definition hta-union :: hta ⇒ hta ⇒ hta
  where hta-union H1 H2 == hta-reindex (hta-union H1 H2)
definition hta-empty :: unit ⇒ hta
  where hta-empty == hta-empty
definition hta-add-qi :: - ⇒ hta ⇒ hta
  where hta-add-qi == hta-add-qi
definition hta-add-rule :: - ⇒ hta ⇒ hta
  where hta-add-rule == hta-add-rule

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

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

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

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

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

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

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

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

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

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

  lemmas htaip-add-qi-correct = hta-add-qi-correct[folded htaip-add-qi-def]

  lemmas htaip-add-rule-correct = hta-add-rule-correct[folded htaip-add-rule-def]

  lemmas htaip-bwd-reduce-correct =
    htaip-bwd-reduce-correct[folded htaip-bwd-reduce-def]

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

  lemmas hta-ensure-idx-f-correct =
    hta-ensure-idx-f-correct[folded hta-ensure-idx-f-def]

  lemmas hta-ensure-idx-s-correct =
    hta-ensure-idx-s-correct[folded hta-ensure-idx-s-def]

  lemmas hta-ensure-idx-sf-correct =
    hta-ensure-idx-sf-correct[folded hta-ensure-idx-sf-def]
end

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

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

lemma hta-union-correct:
assumes \([\text{simp}]: \text{hashedTa} H1 \text{ hashedTa} H2\)
shows
\( \text{ta-lang} \left( \text{hta-\(\alpha\)} \left( \text{htai-union} H1 H2 \right) \right) = \text{ta-lang} \left( \text{hta-\(\alpha\)} H1 \right) \cup \text{ta-lang} \left( \text{hta-\(\alpha\)} H2 \right) \)
\( \text{hashedTa} \left( \text{htai-union} H1 H2 \right) \)
\( \langle \text{proof} \rangle \)

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 hta instead of hta, but have the same semantics otherwise (cf Section 5.10).

5.11.1 Building a Tree Automaton

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

Relevant Lemmas

\( \text{hta-empty-correct}: \text{hta-\(\alpha\)} \left( \text{hta-empty} () \right) = \text{ta-empty} \)
\( \text{hashedTa} \left( \text{hta-empty} () \right) \)
\( \text{ta-empty-lang}: \text{ta-lang} \text{ ta-empty} = \{\} \)
**Function:** hta-add-qi

Adds an initial state to the given automaton.

**Relevant Lemmas**

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

\[
\text{hashedTa } H \implies \text{hashedTa } (\text{hta-add-qi } qi \ H)
\]

**Function:** hta-add-rule

Adds a rule to the given automaton.

**Relevant Lemmas**

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

\[
\text{hashedTa } H \implies \text{hashedTa } (\text{hta-add-rule } r \ H)
\]

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**

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

\[
\text{hashedTa}.\text{hta-mem'}-correct \quad [\text{hashedTa } H; \ \text{hta-has-idx-f } H] \implies \text{hta-mem'} \quad t \ H = (t \in \text{ta-lang} (\text{hta-}\alpha \ H))
\]
Function: \(hta-prod, hta-prod'\)
Compute the product automaton. The computed automaton is in forward-reduced form. The language of the product automaton is the intersection of the languages of the two argument automata.

Relevant Lemmas

\[hta-prod-correct-aux: \quad [hashedTa \ H1 ; \ hashedTa \ H2] \rightarrow hta-\alpha (hta-prod \ H1 \ H2) = ta-fwd-reduce (ta-prod (hta-\alpha H1) (hta-\alpha H2))\]

\[hta-prod-correct: \quad [hashedTa \ H1 ; \ hashedTa \ H2] \rightarrow hta-prod \ (hta-\alpha H1 H2)\]

\[hta-prod'\)-correct-aux: \quad [hashedTa \ H1 ; \ hashedTa \ H2 ; \ hta-has-idx-s \ H1 ; \ hta-has-idx-sf \ H2] \rightarrow hta-\alpha (hta-prod' \ H1 \ H2) = ta-fwd-reduce (ta-prod (hta-\alpha H1) (hta-\alpha H2))\]

\[hta-prod'\)-correct: \quad [hashedTa \ H1 ; \ hashedTa \ H2 ; \ hta-has-idx-s \ H1 ; \ hta-has-idx-sf \ H2] \rightarrow hta-prod' \ (hta-\alpha 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

\[hta-prodWR-correct-aux: \quad [hashedTa \ H1 ; \ hashedTa \ H2] \rightarrow hta-\alpha (hta-prodWR \ H1 \ H2) = ta-prod (hta-\alpha H1) (hta-\alpha H2)\]

\[hta-prodWR-correct: \quad [hashedTa \ H1 ; \ hashedTa \ H2] \rightarrow hta-prodWR \ (hta-\alpha H1 H2)\]

Function: \(hta-union\)
Compute the union of two tree automata.
Relevant Lemmas

\( hta-union-correct' \): \[ \text{hashedTa H}_1; \text{hashedTa H}_2 \] \( \implies \) \( hta-\alpha (hta-union H_1 \ H_2) = \text{ta-union-wrap} (hta-\alpha \ H_1) (hta-\alpha \ H_2) \)

\[ \text{hashedTa H}_1; \text{hashedTa H}_2 \] \( \implies \) \( \text{hashedTa} \ (hta-union H_1 \ H_2) \)

\( hta-union-correct \): \[ \text{hashedTa H}_1; \text{hashedTa H}_2 \] \( \implies \) \( \text{ta-lang} (hta-\alpha (hta-union H_1 \ H_2)) = \text{ta-lang} (hta-\alpha H_1) \cup \text{ta-lang} (hta-\alpha H_2) \)

\[ \text{hashedTa H}_1; \text{hashedTa H}_2 \] \( \implies \) \( \text{hashedTa} \ (hta-union H_1 \ H_2) \)

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

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

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

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

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

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

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

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

5.12 Code Generation

export-code

\begin{verbatim}
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
\end{verbatim}

in SML
module-name Ta

export-code

\begin{verbatim}
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
\end{verbatim}

in Haskell
module-name Ta
(string-classes)

export-code

\begin{verbatim}
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
\end{verbatim}
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\cite{1} model checking algorithm\cite{5} that the author is currently working on.

Refinement The algorithms are first formalized on an abstract level, and then manually refined to become executable. In theory, the abstract algorithms are already executable, as they involve only recursive functions and finite sets. We have experimented with simplifier setups to execute the algorithms in the simplifier, however the performance was quite bad and there where some problems with termination due to the innermost rewriting-strategy used by the simplifier, that required careful crafting of the simplifier setup.

The refinement is done in a somewhat systematic way, using the tools provided by the Isabelle Collections Framework (e.g. a data refinement framework for the while-combinator). However, most of the refinement work is done by hand, and the author believes that it should be possible to do the refinement with more tool support.

Another direction of future work would be to use the tree-automata framework developed here for applications. The author is currently working on a
model-checker for DPNs that uses tree-automata based techniques [5], and plans to use this tree automata framework to generate a verified implementation of this model-checker. However, there are other interesting applications of tree automata, that could be formalized in Isabelle and, using this framework, be refined to efficient executable algorithms.

6.3 Trusted Code Base

In this section we shortly characterize on what our formal proof depends, i.e. how to interpret the information contained in this formal proof and the fact that it is accepted by the Isabelle/HOL system.

First of all, you have to trust the theorem prover and its axiomatization of HOL, the ML-platform, the operating system software and the hardware it runs on. All these components are, in theory, able to cause false theorems to be proven. However, the probability of a false theorem to get proven due to a hardware error or an error in the operating system software is reasonably low. There are errors in hardware and operating systems, but they will usually cause the system to crash or exhibit other unexpected behaviour, instead of causing Isabelle to quietly accept a false theorem and behave normal otherwise. The theorem prover itself is a bit more critical in this aspect. However, Isabelle/HOL is implemented in LCF-style, i.e. all the proofs are eventually checked by a small kernel of trusted code, containing rather simple operations. HOL is the logic that is most frequently used with Isabelle, and it is unlikely that its axiomatization in Isabelle is inconsistent and no one found and reported this inconsistency already.

The next crucial point is the code generator of Isabelle. We derive executable code from our specifications. The code generator contains another (thin) layer of untrusted code. This layer has some known 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 \( f_{\text{div}} :: \text{bool} = \text{while (}\lambda x. \text{True}) \text{id True} \) that is definable in HOL. The lemma \( \forall x. x = \text{if } f_{\text{div}} \text{ then } x \text{ else } x \) is provable in Isabelle and rewriting based on it could, theoretically, be inserted before the code generation process, resulting in code that always diverges.
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


