

Formalization of Dynamic Pushdown Networks in Isabelle/HOL

Peter Lammich

November 20, 2009

Abstract

We present a formalization of Dynamic Pushdown Networks (DPNs) and the automata based algorithm for computing backward reachability sets using Isabelle/HOL. Dynamic pushdown networks are an abstract model for multithreaded, interprocedural programs with dynamic thread creation that was presented by Bouajjani, Mller-Olm and Touili in 2005.

We formalize the notion of a DPN in Isabelle and describe the algorithm for computing the *pre*^{*}-set from a regular set of configurations, and prove its correctness. We first give a nondeterministic description of the algorithm, from that we then infer a deterministic one, from which we can generate executable code using Isabelle's code-generation tool.

Contents

1	Labeled transition systems	3
1.1	Definitions	3
1.2	Basic properties of reflexive-transitive closure	3
2	String rewrite systems	4
2.1	Definitions	4
2.2	Induced Labelled Transition System	5
2.3	Properties of the induced LTS	5
3	Nondeterministic recursive algorithms	6
3.1	Basic properties	6
3.2	Refinement	8
3.3	Extension to reflexive states	10
3.4	Well-foundedness	11
3.4.1	The relations $>$ and \supset on finite domains	12
3.5	Implementation	13

3.5.1	Graphs of functions	13
3.5.2	Deterministic refinement w.r.t. the identity abstraction	14
3.5.3	Recursive characterization	14
4	Finite state machines	15
4.1	Definitions	15
4.2	Basic properties	16
4.3	Reflexive, transitive closure of transition relation	16
4.3.1	Relation of <i>trclAD</i> and <i>trcl</i>	17
4.4	Language of a FSM	18
4.5	Example: Product automaton	18
5	Dynamic pushdown networks	20
5.1	Dynamic pushdown networks	20
5.1.1	Definition	20
5.1.2	Basic properties	21
5.2	M-automata	22
5.2.1	Definition	22
5.2.2	Basic properties	23
5.2.3	Some implications of the M-automata conditions	24
5.3	<i>pre*</i> -sets of regular sets of configurations	27
5.4	Nondeterministic algorithm for <i>pre*</i>	27
5.4.1	Termination	28
5.4.2	Soundness	31
5.4.3	Precision	33
6	Non-executable implementation of the DPN <i>pre*</i>-algorithm	40
6.1	Definitions	41
6.2	Refining <i>ps-R</i>	42
6.3	Termination	43
6.4	Recursive characterization	43
6.5	Correctness	44
7	Tools for executable specifications	44
7.1	Searching in Lists	45
8	Executable algorithms for finite state machines	45
8.1	Word lookup operation	46
8.2	Reachable states and alphabet inferred from transition relation	48
9	Implementation of DPN <i>pre*</i>-algorithm	48
9.1	Representation of DPN and M-automata	49
9.2	Next-element selection	50
9.3	Termination	51
9.3.1	Saturation upper bound	51

9.3.2	Well-foundedness of recursion relation	52
9.3.3	Definition of recursive function	53
9.4	Correctness	53
9.4.1	seln_R refines ps_R	53
9.4.2	Correctness	56

1 Labeled transition systems

```
theory LTS
imports Main
begin
```

Labeled transition systems (LTS) provide a model of a state transition system with named transitions.

1.1 Definitions

A LTS is modelled as a relation, that relates triples of start configuration, transition label and end configuration

```
types ('c,'a) LTS = ('c × 'a × 'c) set
```

Reflexive-transitive closure of LTS

```
inductive-set trcl :: ('c,'a) LTS ⇒ ('c,'a list) LTS for t where
  empty[simp]: (c,[],c) ∈ trcl t |
  cons[simp]: [(c,a,c') ∈ t; (c',w,c'') ∈ trcl t] ⇒ (c,a#w,c'') ∈ trcl t
```

1.2 Basic properties of reflexive-transitive closure

```
lemma trcl-empty-cons[simp]: (c,[],c') ∈ trcl t ⇒ c=c' by (auto elim: trcl.cases)
lemma trcl-single: (c,[a],c') ∈ trcl t ⇒ (c,a,c') ∈ t by (auto elim: trcl.cases)
lemma trcl-uncons: (c,a#w,c') ∈ trcl t ⇒ ∃ ch . (c,a,ch) ∈ t ∧ (ch,w,c') ∈ trcl t
by (auto elim: trcl.cases)
lemma trcl-one-elem: (c,e,c') ∈ t ⇒ (c,[e],c') ∈ trcl t by auto
```

```
lemma trcl-concat: !! c . [(c,w1,c') ∈ trcl t; (c',w2,c'') ∈ trcl t] ⇒ (c,w1@w2,c'') ∈ trcl t
```

```
proof (induct w1)
  case Nil thus ?case by (subgoal-tac c=c') auto
next
  case (Cons a w) thus ?case by (auto dest: trcl-uncons)
qed
```

```
lemma trcl-unconcat: !! c . (c,w1@w2,c') ∈ trcl t ⇒ ∃ ch . (c,w1,ch) ∈ trcl t ∧
(ch,w2,c') ∈ trcl t
proof (induct w1)
  case Nil hence (c,[],c) ∈ trcl t ∧ (c,w2,c') ∈ trcl t by auto
  thus ?case by fast
```

next
case (*Cons a w1*) **note** *IHP = this*
hence $(c, a \# (w1 @ w2), c') \in \text{trcl } t$ **by** *simp*
with *trcl-uncons* **obtain** *chh* **where** $(c, a, chh) \in t \wedge (chh, w1 @ w2, c') \in \text{trcl } t$ **by**
fast
moreover with *IHP* **obtain** *ch* **where** $(chh, w1, ch) \in \text{trcl } t \wedge (ch, w2, c') \in \text{trcl } t$
by *fast*
ultimately have $(c, a \# w1, ch) \in \text{trcl } t \wedge (ch, w2, c') \in \text{trcl } t$ **by** *auto*
thus *?case* **by** *fast*
qed

lemma *trcl-mono*: $A \subseteq B \implies \text{trcl } A \subseteq \text{trcl } B$
apply (*clarsimp*)
apply (*erule trcl.induct*)
apply *auto*
done

lemma *trcl-struct*: $\llbracket (s, w, s') \in \text{trcl } D; D \subseteq S \times A \times S' \rrbracket \implies (s = s' \wedge w = []) \vee (s \in S \wedge s' \in S' \wedge w \in \text{lists } A)$ **proof** (*induct w, simp-all*)
fix *e w s*
assume *IH*: $\llbracket sh, w, s' \rrbracket \in \text{trcl } D \implies sh = s' \wedge w = [] \vee sh \in S \wedge s' \in S' \wedge w \in \text{lists } A$
assume *A*: $(s, e \# w, s') \in \text{trcl } D \wedge D \subseteq S \times A \times S'$
then obtain *sh* **where** *SPLIT*: $(s, e, sh) \in D \wedge (sh, w, s') \in \text{trcl } D$ **by** (*blast dest: trcl-uncons*)
from *SPLIT A* **have** $s \in S \wedge e \in A \wedge sh \in S'$ **by** *auto*
moreover from *SPLIT IH* **have** $(sh = s' \wedge w = []) \vee (s' \in S' \wedge w \in \text{lists } A)$ **by** *blast*
ultimately show $s \in S \wedge s' \in S' \wedge e \# w \in \text{lists } A$ **by** *auto*
qed

lemma *trcl-structE*: $\llbracket (s, w, s') \in \text{trcl } D; D \subseteq S \times A \times S'; \llbracket s = s'; w = [] \rrbracket \implies P; \llbracket s \in S; s' \in S'; w \in \text{lists } A \rrbracket \implies P \rrbracket \implies P$
by (*blast dest: trcl-struct*)

end

2 String rewrite systems

theory *SRS*
imports *Main LTS*
begin

This formalizes systems of labelled string rewrite rules and the labelled transition systems induced by them. DPNs are special string rewrite systems.

2.1 Definitions

types

$(\prime c, \prime l)$ *rewrite-rule* = $\prime c \text{ list} \times \prime l \times \prime c \text{ list}$
 $(\prime c, \prime l)$ *SRS* = $(\prime c, \prime l)$ *rewrite-rule set*

syntax

syn-rew-rule :: $\prime c \text{ list} \Rightarrow \prime l \Rightarrow \prime c \text{ list} \Rightarrow (\prime c, \prime l)$ *rewrite-rule* ($- \hookrightarrow_-$ - [51,51,51] 51)

translations

$s \hookrightarrow_a s' \Rightarrow (s, a, s')$

A (labelled) rewrite rule (s, a, s') consists of the left side s , the label a and the right side s' . Intuitively, it means that a substring s can be rewritten to s' by an a -step. A string rewrite system is a set of labelled rewrite rules

2.2 Induced Labelled Transition System

A string rewrite systems induces a labelled transition system on strings by rewriting substrings according to the rules

inductive-set $tr :: (\prime c, \prime l)$ *SRS* $\Rightarrow (\prime c \text{ list}, \prime l)$ *LTS for S*

where

rewrite: $(s \hookrightarrow_a s') \in S \Rightarrow (ep@s@es, a, ep@s'@es) \in tr S$

2.3 Properties of the induced LTS

Adding characters at the start or end of a state does not influence the capability of making a transition

lemma *srs-ext-s*: $(s, a, s') \in tr S \Rightarrow (wp@s@ws, a, wp@s'@ws) \in tr S$ **proof** –

assume $(s, a, s') \in tr S$

then obtain $ep \ es \ r \ r'$ **where** $s = ep@r@es \wedge s' = ep@r'@es \wedge (r, a, r') \in S$ **by**
(fast elim: tr.cases)

moreover hence $((wp@ep)@r@(es@ws), a, (wp@ep)@r'@(es@ws)) \in tr S$ **by**
(fast intro: tr.rewrite)

ultimately show *?thesis* **by** *auto*

qed

lemma *srs-ext-both*: $(s, w, s') \in trcl (tr S) \Rightarrow (wp@s@ws, w, wp@s'@ws) \in trcl (tr S)$

apply *(induct s w s' rule: trcl.induct)*

apply *(simp)*

apply *(subgoal-tac wp @ c @ ws \hookrightarrow_a wp @ c' @ ws $\in tr S$)*

apply *(auto intro: srs-ext-s)*

done

corollary *srs-ext-cons*: $(s, w, s') \in trcl (tr S) \Rightarrow (e\#s, w, e\#s') \in trcl (tr S)$ **by** *(rule srs-ext-both[where wp=[e] and ws=[], simplified])*

corollary *srs-ext-pre*: $(s, w, s') \in trcl (tr S) \Rightarrow (wp@s, w, wp@s') \in trcl (tr S)$ **by**
(rule srs-ext-both[where ws=[], simplified])

corollary *srs-ext-post*: $(s, w, s') \in trcl (tr S) \Rightarrow (s@ws, w, s'@ws) \in trcl (tr S)$ **by**
(rule srs-ext-both[where wp=[], simplified])

lemmas *srs-ext* = *srs-ext-both* *srs-ext-pre* *srs-ext-post*

end

3 Nondeterministic recursive algorithms

theory *NDET*
imports *Main*
begin

This theory models nondeterministic, recursive algorithms by means of a step relation.

An algorithm is modelled as follows:

1. Start with some state s
2. If there is no s' with $(s, s') \in R$, terminate with state s
3. Else set $s := s'$ and continue with step 2

Thus, R is the step relation, relating the previous with the next state. If the state is not in the domain of R , the algorithm terminates.

The relation $A\text{-rel } R$ describes the non-reflexive part of the algorithm, that is all possible mappings for non-terminating initial states. We will first explore properties of this non-reflexive part, and then transfer them to the whole algorithm, that also specifies how terminating initial states are treated.

inductive-set $A\text{-rel} :: ('s \times 's) \text{ set} \Rightarrow ('s \times 's) \text{ set}$ **for** R

where

$A\text{-rel-base}$: $\llbracket (s, s') \in R; s' \notin \text{Domain } R \rrbracket \Longrightarrow (s, s') \in A\text{-rel } R$ |

$A\text{-rel-step}$: $\llbracket (s, sh) \in R; (sh, s') \in A\text{-rel } R \rrbracket \Longrightarrow (s, s') \in A\text{-rel } R$

3.1 Basic properties

The algorithm just terminates at terminating states

lemma *termstate*: $(s, s') \in A\text{-rel } R \Longrightarrow s' \notin \text{Domain } R$ **by** (*induct rule*: $A\text{-rel.induct}$, *auto*)

lemma *dom-subset*: $\text{Domain } (A\text{-rel } R) \subseteq \text{Domain } R$ **by** (*unfold Domain-def*) (*auto elim*: $A\text{-rel.induct}$)

We can use invariants to reason over properties of the algorithm

constdefs

is-inv R $s0$ $P == P s0 \wedge (\forall s s'. (s, s') \in R \wedge P s \longrightarrow P s')$

lemma *inv*: $\llbracket (s0, sf) \in A\text{-rel } R; \text{is-inv } R \ s0 \ P \rrbracket \Longrightarrow P \ sf$ **by** (*unfold is-inv-def*, *induct rule: A-rel.induct*) *blast+*

lemma *invI*: $\llbracket P \ s0; !! \ s \ s'. \llbracket (s, s') \in R; P \ s \rrbracket \Longrightarrow P \ s \rrbracket \Longrightarrow \text{is-inv } R \ s0 \ P$ **by** (*unfold is-inv-def*, *blast*)

lemma *inv2*: $\llbracket (s0, sf) \in A\text{-rel } R; P \ s0; !! \ s \ s'. \llbracket (s, s') \in R; P \ s \rrbracket \Longrightarrow P \ s \rrbracket \Longrightarrow P \ sf$
apply (*subgoal-tac is-inv R s0 P*)
apply (*blast intro: inv*)
apply (*blast intro: invI*)
done

To establish new invariants, we can use already existing invariants

lemma *inv-useI*: $\llbracket P \ s0; !! \ s \ s'. \llbracket (s, s') \in R; P \ s; !! \ P'. \text{is-inv } R \ s0 \ P' \Longrightarrow P' \ s \rrbracket \Longrightarrow P \ s' \rrbracket \Longrightarrow \text{is-inv } R \ s0 \ (\lambda s. P \ s \wedge (\forall P'. \text{is-inv } R \ s0 \ P' \longrightarrow P' \ s))$
apply (*rule invI*)
apply (*simp (no-asm) only: is-inv-def, blast*)
apply *safe*
apply *blast*
apply (*subgoal-tac P' s*)
apply (*simp (no-asm-use) only: is-inv-def, blast*)
apply *fast*
done

If the inverse step relation is well-founded, the algorithm will terminate for every state in *Domain R* (\subseteq -direction). The \supseteq -direction is from *dom-subset*

lemma *wf-dom-eq*: $wf \ (R^{-1}) \Longrightarrow \text{Domain } R = \text{Domain } (A\text{-rel } R)$ **proof** –
assume *WF*: $wf \ (R^{-1})$
hence [*rule-format*]: $!!s. \forall s'. (s, s') \in R \longrightarrow (\exists sf. (s, sf) \in A\text{-rel } R)$
apply –
proof (*erule wf-induct-rule, auto*)
fix *x s'*
assume *IH*: $!!y. (x, y) \in R \Longrightarrow (\exists s'. (y, s') \in R) \longrightarrow (\exists sf. (y, sf) \in A\text{-rel } R)$ **and**
A: $(x, s') \in R$
{
assume $s' \notin \text{Domain } R$
with *A* **have** $(x, s') \in A\text{-rel } R$ **by** (*blast intro: A-rel-base*)
} **moreover** **{**
assume $s' \in \text{Domain } R$
then obtain *st* **where** $(s', st) \in R$ **by** (*unfold Domain-def, auto*)
with *A* **obtain** *sf* **where** $(s', sf) \in A\text{-rel } R$ **by** *blast*
with *A* **have** $(x, sf) \in A\text{-rel } R$ **by** (*blast intro: A-rel-step*)
hence $\exists sf. (x, sf) \in A\text{-rel } R$ **by** *blast*
} **ultimately show** $\exists sf. (x, sf) \in A\text{-rel } R$ **by** *blast*
qed
hence $\text{Domain } R \subseteq \text{Domain } (A\text{-rel } R)$ **by** (*unfold Domain-def, auto*)
with *dom-subset* **show** *?thesis* **by** *force*
qed

3.2 Refinement

Refinement is a simulation property between step relations.

We define refinement w.r.t. an abstraction relation α , that relates abstract to concrete states. The refining step-relation is called more concrete than the refined one.

constdefs

$refines :: ('s*'s) set \Rightarrow ('r*'s) set \Rightarrow ('r*'r) set \Rightarrow bool$ ($- \leq -$ [50,50,50] 50)
 $R \leq_{\alpha} S == R \circ \alpha \subseteq \alpha \circ S \wedge \alpha \text{ `` Domain } S \subseteq \text{Domain } R$

lemma *refinesI*: $\llbracket R \circ \alpha \subseteq \alpha \circ S; \alpha \text{ `` Domain } S \subseteq \text{Domain } R \rrbracket \Longrightarrow R \leq_{\alpha} S$ **by** (*unfold refines-def, auto*)

lemma *refinesE*: $R \leq_{\alpha} S \Longrightarrow R \circ \alpha \subseteq \alpha \circ S$

$R \leq_{\alpha} S \Longrightarrow \alpha \text{ `` Domain } S \subseteq \text{Domain } R$

by (*unfold refines-def, auto*)

Intuitively, the first condition for refinement means, that for each concrete step $(c, c') \in S$ where the start state c has an abstract counterpart $(a, c) \in \alpha$, there is also an abstract counterpart of the end state $(a', c') \in \alpha$ and the step can also be done on the abstract counterparts $(a, a') \in R$.

lemma *refines-compI*:

assumes $A: !! a c c'. \llbracket (a, c) \in \alpha; (c, c') \in S \rrbracket \Longrightarrow \exists a'. (a, a') \in R \wedge (a', c') \in \alpha$

shows $S \circ \alpha \subseteq \alpha \circ R$ **using** A **by** (*auto simp add: rel-comp-def*)

lemma *refines-compE*: $\llbracket S \circ \alpha \subseteq \alpha \circ R; (a, c) \in \alpha; (c, c') \in S \rrbracket \Longrightarrow \exists a'. (a, a') \in R \wedge (a', c') \in \alpha$ **by** (*auto simp add: rel-comp-def*)

Intuitively, the second condition for refinement means, that if there is an abstract step $(a, a') \in R$, where the start state has a concrete counterpart c , then there must also be a concrete step from c . Note that this concrete step is not required to lead to the concrete counterpart of a' . In fact, it is only important that there is such a concrete step, ensuring that the concrete algorithm will not terminate on states on that the abstract algorithm continues execution.

lemma *refines-domI*:

assumes $A: !! a a' c. \llbracket (a, c) \in \alpha; (a, a') \in R \rrbracket \Longrightarrow c \in \text{Domain } S$

shows $\alpha \text{ `` Domain } R \subseteq \text{Domain } S$ **using** A **by** *auto*

lemma *refines-domE*: $\llbracket \alpha \text{ `` Domain } R \subseteq \text{Domain } S; (a, c) \in \alpha; (a, a') \in R \rrbracket \Longrightarrow c \in \text{Domain } S$ **by** *auto*

lemma *refinesI2*:

assumes $A: !! a c c'. \llbracket (a, c) \in \alpha; (c, c') \in S \rrbracket \Longrightarrow \exists a'. (a, a') \in R \wedge (a', c') \in \alpha$

assumes $B: !! a a' c. \llbracket (a, c) \in \alpha; (a, a') \in R \rrbracket \Longrightarrow c \in \text{Domain } S$

shows $S \leq_{\alpha} R$ **by** (*simp only: refinesI A refines-compI B refines-domI*)

lemma *refinesE2*:

$\llbracket S \leq_{\alpha} R; (a,c) \in \alpha; (c,c') \in S \rrbracket \implies \exists a'. (a,a') \in R \wedge (a',c') \in \alpha$
 $\llbracket S \leq_{\alpha} R; (a,c) \in \alpha; (a,a') \in R \rrbracket \implies c \in \text{Domain } S$
by (*blast dest: refinesE refines-compE refines-domE*)⁺

Reflexivity of identity refinement

lemma *refines-id-refl*[*intro!*, *simp*]: $R \leq_{Id} R$ **by** (*auto intro: refinesI*)

Transitivity of refinement

lemma *refines-trans*: **assumes** $R: R \leq_{\alpha} S \quad S \leq_{\beta} T$ **shows** $R \leq_{\alpha \circ \beta} T$

proof (*rule refinesI*)

$\{$
 fix $s \ s' \ t'$
 assume $A: (s,s') \in \alpha \circ \beta \ (s',t') \in R$
 then obtain sh **where** $(s,sh) \in \beta \wedge (sh,s') \in \alpha$ **by** (*blast*)
 with $A \ R$ **obtain** $t \ th$ **where** $(sh,th) \in S \wedge (th,t') \in \alpha \wedge (s,t) \in T \wedge (t,th) \in \beta$ **by**
 (*blast dest: refinesE*)
 hence $(s,t') \in (\alpha \circ \beta) \circ T$ **by** *blast*
 $\}$ **thus** $R \circ (\alpha \circ \beta) \subseteq (\alpha \circ \beta) \circ T$ **by** *blast*

next

$\{$
 fix $s \ s'$
 assume $A: s \in \text{Domain } T \ (s,s') \in \alpha \circ \beta$
 then obtain sh **where** $(s,sh) \in \beta \wedge (sh,s') \in \alpha$ **by** *blast*

with $R \ A$ **have** $s' \in \text{Domain } R$ **by** (*blast dest!: refinesE*)

thus $(\alpha \circ \beta) \subseteq \text{Domain } T \subseteq \text{Domain } R$ **by** (*unfold Domain-def, blast*)

qed

Property transfer lemma

lemma *refines-A-rel*[*rule-format*]:

assumes $R: R \leq_{\alpha} S$ **and** $A: (r,r') \in A\text{-rel } R$

shows $\forall s. (s,r) \in \alpha \implies (\exists s'. (s',r') \in \alpha \wedge (s,s') \in A\text{-rel } S)$

using A

proof (*induct rule: A-rel.induct, auto*)

fix $r \ r' \ s$

assume $C: (r,r') \in R \ r' \notin \text{Domain } R \ (s,r) \in \alpha$

with R **obtain** s' **where** $(s,s') \in S \wedge (s',r') \in \alpha \wedge s' \notin \text{Domain } S$ **by** (*blast dest: refinesE*)

hence $(s',r') \in \alpha \wedge (s,s') \in A\text{-rel } S$ **by** (*blast intro: A-rel-base*)

thus $\exists s'. (s',r') \in \alpha \wedge (s,s') \in A\text{-rel } S$ **by** (*blast*)

next

fix $r \ rh \ r' \ s$

assume $A: (r,rh) \in R \ (rh,r') \in A\text{-rel } R \ (s,r) \in \alpha$ **and** $IH: \forall sh. (sh, rh) \in \alpha \implies (\exists s'. (s', rh) \in \alpha \wedge (sh, s') \in A\text{-rel } S)$

with R **obtain** sh **where** $STEP: (sh,rh) \in \alpha \wedge (s,sh) \in S$ **by** (*blast dest: refinesE*)

with IH **obtain** s' **where** $(s',rh) \in \alpha \wedge (sh,s') \in A\text{-rel } S$ **by** *blast*

with $STEP$ **have** $(s', rh) \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (*blast intro: A-rel-step*)

thus $\exists s'. (s', rh) \in \alpha \wedge (s, s') \in A\text{-rel } S$ **by** (*blast*)

qed

Property transfer lemma for single-valued abstractions (i.e. abstraction functions)

lemma *refines-A-rel-sv*: $\llbracket R \leq_{\alpha} S; (r, r') \in A\text{-rel } R; \text{single-valued } (\alpha^{-1}); (s, r) \in \alpha; (s', r') \in \alpha \rrbracket \implies (s, s') \in A\text{-rel } S$ **by** (*blast dest: single-valuedD refines-A-rel*)

3.3 Extension to reflexive states

Up to now we only defined how to relate initial states to terminating states if the algorithm makes at least one step. In this section, we also add the reflexive part: Initial states for that no steps can be made are mapped to themselves.

constdefs

ndet-algo $R == (A\text{-rel } R) \cup \{(s, s) \mid s. s \notin \text{Domain } R\}$

lemma *ndet-algo-A-rel*: $\llbracket x \in \text{Domain } R; (x, y) \in \text{ndet-algo } R \rrbracket \implies (x, y) \in A\text{-rel } R$ **by** (*unfold ndet-algo-def*) *auto*

lemma *ndet-algoE*: $\llbracket (s, s') \in \text{ndet-algo } R; \llbracket (s, s') \in A\text{-rel } R \rrbracket \implies P; \llbracket s = s'; s \notin \text{Domain } R \rrbracket \implies P \rrbracket \implies P$ **by** (*unfold ndet-algo-def, auto*)

lemma *ndet-algoE'*: $\llbracket (s, s') \in \text{ndet-algo } R; \llbracket (s, s') \in A\text{-rel } R; s \in \text{Domain } R; s' \notin \text{Domain } R \rrbracket \implies P; \llbracket s = s'; s \notin \text{Domain } R \rrbracket \implies P \rrbracket \implies P$

using *dom-subset[of R] termstate[of s s' R]*

by (*auto elim!: ndet-algoE*)

ndet-algo is total (i.e. the algorithm is defined for every initial state), if R^{-1} is well founded

lemma *ndet-algo-total*: $wf (R^{-1}) \implies \text{Domain } (\text{ndet-algo } R) = UNIV$ **by** (*unfold ndet-algo-def*) (*auto simp add: wf-dom-eq*)

The result of the algorithm is always a terminating state

lemma *termstate-ndet-algo*: $(s, s') \in \text{ndet-algo } R \implies s' \notin \text{Domain } R$ **by** (*unfold ndet-algo-def, auto dest: termstate*)

Property transfer lemma for *ndet-algo*

lemma *refines-ndet-algo[rule-format]*:

assumes $R: S \leq_{\alpha} R$ **and** $A: (c, c') \in \text{ndet-algo } S$

shows $\forall a. (a, c) \in \alpha \longrightarrow (\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R)$

proof (*intro allI impI*)

fix a **assume** $B: (a, c) \in \alpha$

{ **assume** *CASE*: $c \in \text{Domain } S$

with A **have** $(c, c') \in A\text{-rel } S$ **by** (*blast elim: ndet-algoE*)

with R B **obtain** a' **where** $(a', c') \in \alpha \wedge (a, a') \in A\text{-rel } R$ **by** (*blast dest:*

refines-A-rel)

moreover hence $(a, a') \in \text{ndet-algo } R$ **by** (*unfold ndet-algo-def, simp*)

ultimately have $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ **by** *blast*

} **moreover** {

assume *CASE*: $c \notin \text{Domain } S$

with A **have** $c=c'$ **by** (*blast elim: ndet-algoE'*)
moreover have $a \notin \text{Domain } R$ **proof** (*rule ccontr, simp*)
assume $a \in \text{Domain } R$
with B R **have** $c \in \text{Domain } S$ **by** (*auto elim: refinesE2*)
with $CASE$ **show** $False$..
qed
ultimately have $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ **using** B **by** (*unfold ndet-algo-def, blast*)
} ultimately show $\exists a'. (a', c') \in \alpha \wedge (a, a') \in \text{ndet-algo } R$ **by** *blast*
qed

Property transfer lemma for single-valued abstractions (i.e. Abstraction functions)

lemma *refines-ndet-algo-sv*: $\llbracket S \leq_{\alpha} R; (c, c') \in \text{ndet-algo } S; \text{single-valued } (\alpha^{-1}); (a, c) \in \alpha; (a', c') \in \alpha \rrbracket \implies (a, a') \in \text{ndet-algo } R$ **by** (*blast dest: single-valuedD refines-ndet-algo*)

3.4 Well-foundedness

lemma *wf-imp-minimal*: $\llbracket wf \ S; x \in Q \rrbracket \implies \exists z \in Q. (\forall x. (x, z) \in S \longrightarrow x \notin Q)$ **by** (*auto iff add: wf-eq-minimal*)

This lemma allows to show well-foundedness of a refining relation by providing a well-founded refined relation for each element in the domain of the refining relation.

lemma *refines-wf*:

assumes $A: !!r. \llbracket r \in \text{Domain } R \rrbracket \implies (s \ r, r) \in \alpha \ r \wedge R \leq_{\alpha} \ r \ S \ r \wedge wf \ ((S \ r)^{-1})$
shows $wf \ (R^{-1})$

proof (*auto simp add: wf-eq-minimal*)

case (*goal1 Q e*) **note** $NOTEMPTY = \text{this}$

moreover {

assume $e \notin \text{Domain } R$

hence $\forall y. (e, y) \in R \longrightarrow y \notin Q$ **by** *blast*

} moreover {

assume $C: e \in \text{Domain } R$

with A **have** $MAP: (s \ e, e) \in \alpha \ e$ **and** $REF: R \leq_{\alpha} \ e \ S \ e$ **and** $WF: wf \ ((S \ e)^{-1})$

by (*auto*)

let $?aQ = ((\alpha \ e)^{-1}) \ \text{“ } Q$

from $MAP \ NOTEMPTY$ **have** $s \ e \in ?aQ$ **by** *auto*

with WF *wf-imp-minimal*[*of* $(S \ e)^{-1}$, *simplified*] **have** $\exists z \in ?aQ. (\forall x. (z, x) \in S \ e \longrightarrow x \notin ?aQ)$ **by** *auto*

then obtain z **where** $ZMIN: z \in ?aQ \wedge (\forall x. (z, x) \in S \ e \longrightarrow x \notin ?aQ)$ **by** *blast*

then obtain q **where** $QP: (z, q) \in \alpha \ e \wedge q \in Q$ **by** *blast*

have $\forall x. (q, x) \in R \longrightarrow x \notin Q$ **proof** (*intro allI impI*)

fix x

assume $(q, x) \in R$

with $REF \ QP$ **obtain** xt **where** $ZREF: (z, xt) \in S \ e \wedge (xt, x) \in \alpha \ e$ **by** (*blast dest: refinesE*)

with $ZMIN$ **have** $xt \notin ?aQ$ **by** *simp*

moreover from $ZREF$ **have** $x \in Q \implies xt \in ?aQ$ **by** *blast*

```

    ultimately show  $x \notin Q$  by blast
  qed
  with  $QP$  have  $\exists q \in Q. \forall y. (q, y) \in R \longrightarrow y \notin Q$  by blast
} ultimately show  $\exists z \in Q. \forall y. (z, y) \in R \longrightarrow y \notin Q$  by blast
qed

```

3.4.1 The relations $>$ and \supset on finite domains

constdefs

```

greaterN N == {(i,j) . j < i & i ≤ (N::nat)}
greaterS S == {(a,b) . b ⊂ a & a ⊆ (S::'a set)}

```

$>$ on initial segment of nat is well founded

```

lemma wf-greaterN: wf (greaterN N)
apply (unfold greaterN-def)
apply (rule wf-subset[of measure (λk. (N-k))], blast)
apply (clarify, simp add: measure-def inv-image-def)
done

```

Strict version of *card-mono*

```

lemma card-mono-strict: [finite B; A ⊂ B] ⇒ card A < card B proof –
  assume F: finite B and S: A ⊂ B
  hence FA: finite A by (auto intro: finite-subset)
  from S obtain x where P: x ∈ B ∧ x ∉ A ∧ A – {x} = A ∧ insert x A ⊆ B by
  auto
  with FA card-insert[of A x] have card (insert x A) = Suc (card A) by (simp)
  moreover from F P have card (insert x A) ≤ card B by (fast intro: card-mono)
  ultimately show ?thesis by simp
qed

```

\supset on finite sets is well founded

This is shown here by embedding the \supset relation into the $>$ relation, using cardinality

```

lemma wf-greaterS[redef-wf]: finite S ⇒ wf (greaterS S) proof –
  assume FS: finite S — For this purpose, we show that we can embed greaterS
  into the greaterN by the inverse image of cardinality
  have {(a,b) . b ⊂ a ∧ a ⊆ S} ⊆ inv-image (greaterN (card S)) card proof (simp
  add: inv-image-def greaterN-def)
  {
    fix a b
    assume A: b ⊂ a a ⊆ S
    with FS have Fab: finite a finite b by (auto simp add: finite-subset)
    with A FS have card b < card a & card a ≤ card S by (fast intro: card-mono
  card-mono-strict)
  } note R=this
  thus {(a, b) . b ⊂ a ∧ a ⊆ S} ⊆ {(x, y) . card y < card x ∧ card x ≤ card S}
by blast
qed

```

thus *?thesis* **by** (*unfold greaterS-def, blast intro: wf-greaterN wf-subset*)
qed

This lemma shows well-foundedness of saturation algorithms, where in each step some set is increased, and this set remains below some finite upper bound

lemma *sat-wf*:

assumes *subset*: $\forall r r'. (r, r') \in R \implies \alpha r \subseteq \alpha r' \wedge \alpha r' \subseteq U$

assumes *finite*: *finite U*

shows *wf (R⁻¹)*

proof –

have $R^{-1} \subseteq \text{inv-image } (\text{greaterS } U) \alpha$ **by** (*auto simp add: inv-image-def greaterS-def dest: subset*)

moreover have *wf (inv-image (greaterS U) α)* **using** *finite* **by** (*blast intro: wf-greaterS*)

ultimately show *?thesis* **by** (*blast intro: wf-subset*)

qed

3.5 Implementation

The first step to implement a nondeterministic algorithm specified by a relation R is to provide a deterministic refinement w.r.t. the identity abstraction Id . We can describe such a deterministic refinement as the graph of a partial function sel . We call this function a selector function, because it selects the next state from the possible states specified by R .

In order to get a working implementation, we must prove termination. That is, we have to show that $(\text{graph } sel)^{-1}$ is well-founded. If we already know that R^{-1} is well-founded, this property transfers to $(\text{graph } sel)^{-1}$.

Once obtained well-foundedness, we can use the selector function to implement the following recursive function:

algo s = case sel s of None \Rightarrow s | Some s' \Rightarrow algo s'

And we can show, that *algo* is consistent with *ndet-algo R*, that is $(s, \text{algo } s) \in \text{ndet-algo } R$.

3.5.1 Graphs of functions

The graph of a (partial) function is the relation of arguments and function values

constdefs

graph f == {(x, x') . f x = Some x'}

lemma *graphI[intro]*: $f x = \text{Some } x' \implies (x, x') \in \text{graph } f$ **by** (*unfold graph-def, auto*)

lemma *graphD[dest]*: $(x, x') \in \text{graph } f \implies f x = \text{Some } x'$ **by** (*unfold graph-def, auto*)

lemma *graph-dom-iff1*: $(x \notin \text{Domain } (\text{graph } f)) = (f x = \text{None})$ **by** (*cases f x*) *auto*
lemma *graph-dom-iff2*: $(x \in \text{Domain } (\text{graph } f)) = (f x \neq \text{None})$ **by** (*cases f x*) *auto*

3.5.2 Deterministic refinement w.r.t. the identity abstraction

lemma *detRef-eq*: $(\text{graph } \text{sel} \leq_{\text{Id}} R) = ((\forall s s'. \text{sel } s = \text{Some } s' \longrightarrow (s, s') \in R) \wedge (\forall s. \text{sel } s = \text{None} \longrightarrow s \notin \text{Domain } R))$
by (*unfold refines-def*) (*auto iff add: graph-dom-iff2*)

lemma *detRef-wf-transfer*: $\llbracket \text{wf } (R^{-1}); \text{graph } \text{sel} \leq_{\text{Id}} R \rrbracket \Longrightarrow \text{wf } ((\text{graph } \text{sel})^{-1})$
by (*rule refines-wf* [**where** $s = \text{id}$ **and** $\alpha = \lambda x. \text{Id}$ **and** $S = \lambda x. R$]) *simp*

3.5.3 Recursive characterization

locale *detRef-impl* =
fixes *algo* **and** *sel* **and** *R*
assumes *detRef*: $\text{graph } \text{sel} \leq_{\text{Id}} R$
assumes *algo-rec*[*simp*]: $!! s s'. \text{sel } s = \text{Some } s' \Longrightarrow \text{algo } s = \text{algo } s'$ **and**
algo-term[*simp*]: $!! s. \text{sel } s = \text{None} \Longrightarrow \text{algo } s = s$
assumes *wf*: $\text{wf } ((\text{graph } \text{sel})^{-1})$

lemma (**in** *detRef-impl*) *sel-cons*:
 $\text{sel } s = \text{Some } s' \Longrightarrow (s, s') \in R$
 $\text{sel } s = \text{None} \Longrightarrow s \notin \text{Domain } R$
 $s \in \text{Domain } R \Longrightarrow \exists s'. \text{sel } s = \text{Some } s'$
 $s \notin \text{Domain } R \Longrightarrow \text{sel } s = \text{None}$
using *detRef*
by (*simp-all only: detRef-eq*) (*cases sel s, blast, blast*)**+**

lemma (**in** *detRef-impl*) *algo-correct*: $(s, \text{algo } s) \in \text{ndet-algo } R$ **proof** –
{
assume *C*: $s \in \text{Domain } R$
have $!!s. s \in \text{Domain } R \longrightarrow (s, \text{algo } s) \in A\text{-rel } R$
apply (*rule wf-induct*[*of* $(\text{graph } \text{sel})^{-1} \lambda s. s \in \text{Domain } R \longrightarrow (s, \text{algo } s) \in A\text{-rel } R$])
apply (*rule wf*)
apply (*intro impI*)
apply (*simp only: converse-iff*)
proof –
fix *s*
assume *A*: $s \in \text{Domain } R$ **and** *IH*: $\forall y. (s, y) \in \text{graph } \text{sel} \longrightarrow y \in \text{Domain } R \longrightarrow (y, \text{algo } y) \in A\text{-rel } R$
then obtain *sh* **where** *SH*: $\text{sel } s = \text{Some } sh \wedge (s, sh) \in R$ **using** *sel-cons* **by**
blast
hence *AS*: $\text{algo } s = \text{algo } sh$ **by** *auto*
{
assume *C*: $sh \notin \text{Domain } R$
hence $\text{sel } sh = \text{None}$ **by** (*auto dest: sel-cons*)
hence $\text{algo } sh = sh$ **by** (*auto*)
moreover from *SH C* **have** $(s, sh) \in A\text{-rel } R$ **by** (*blast intro: A-rel-base*)
}

```

    ultimately have  $(s, \text{algo } s) \in A\text{-rel } R$  using AS by simp
  } moreover {
    assume  $C: sh \in \text{Domain } R$ 
    with SH IH AS A have  $(sh, \text{algo } s) \in A\text{-rel } R$  by auto
    with SH have  $(s, \text{algo } s) \in A\text{-rel } R$  by (blast intro: A-rel-step)
  } ultimately show  $(s, \text{algo } s) \in A\text{-rel } R$  by blast
qed
with C have  $(s, \text{algo } s) \in A\text{-rel } R$  by simp
hence ?thesis by (unfold ndet-algo-def, auto)
} moreover {
  assume  $C: s \notin \text{Domain } R$ 
  hence  $s = \text{algo } s$  by (auto dest: sel-cons)
  with C have ?thesis by (unfold ndet-algo-def, auto)
} ultimately show ?thesis by blast
qed

```

end

4 Finite state machines

```

theory FSM
imports Main LTS
begin

```

This theory models nondeterministic finite state machines with explicit set of states and alphabet. ε -transitions are not supported.

4.1 Definitions

```

record ('s, 'a) FSM-rec =
  Q :: 's set — The set of states
  Σ :: 'a set — The alphabet
  δ :: ('s, 'a) LTS — The transition relation
  s0 :: 's — The initial state
  F :: 's set — The set of final states

```

```

locale FSM =
  fixes A
  assumes delta-cons:  $(q, l, q') \in \delta \ A \implies q \in Q \ A \wedge l \in \Sigma \ A \wedge q' \in Q \ A$  — The transition relation is consistent with the set of states and the alphabet
  assumes s0-cons:  $s0 \ A \in Q \ A$  — The initial state is a state
  assumes F-cons:  $F \ A \subseteq Q \ A$  — The final states are states
  assumes finite-states: finite (Q A) — The set of states is finite
  assumes finite-alphabet: finite (Σ A) — The alphabet is finite

```

4.2 Basic properties

lemma (in *FSM*) *finite-delta-dom*: $finite (Q A \times \Sigma A \times Q A)$ **proof** –
from *finite-states finite-alphabet finite-cartesian-product*[of $\Sigma A Q A$] **have** $finite (\Sigma A \times Q A)$ **by** *fast*
with *finite-states finite-cartesian-product*[of $Q A \Sigma A \times Q A$] **show** $finite (Q A \times \Sigma A \times Q A)$ **by** *fast*
qed

lemma (in *FSM*) *finite-delta*: $finite (\delta A)$ **proof** –
have $\delta A \subseteq Q A \times \Sigma A \times Q A$ **by** (*auto simp add: delta-cons*)
with *finite-delta-dom* **show** *?thesis* **by** (*simp add: finite-subset*)
qed

4.3 Reflexive, transitive closure of transition relation

Reflexive transitive closure on restricted domain

inductive-set *trclAD* :: $(s, 'a, 'c)$ *FSM-rec-scheme* $\Rightarrow (s, 'a)$ *LTS* $\Rightarrow (s, 'a)$ *list*
LTS
for $A D$
where
empty[*simp*]: $s \in Q A \Longrightarrow (s, [], s) \in trclAD A D$ |
cons[*simp*]: $\llbracket (s, e, s') \in D; s \in Q A; e \in \Sigma A; (s', w, s'') \in trclAD A D \rrbracket \Longrightarrow (s, e \# w, s'') \in trclAD A D$

abbreviation $trclA A == trclAD A (\delta A)$

lemma *trclAD-empty-cons*[*simp*]: $(c, [], c') \in trclAD A D \Longrightarrow c = c'$ **by** (*auto elim: trclAD.cases*)

lemma *trclAD-single*: $(c, [a], c') \in trclAD A D \Longrightarrow (c, a, c') \in D$ **by** (*auto elim: trclAD.cases*)

lemma *trclAD-elems*: $(c, w, c') \in trclAD A D \Longrightarrow c \in Q A \wedge w \in lists (\Sigma A) \wedge c' \in Q A$ **by** (*erule trclAD.induct, auto*)

lemma *trclAD-one-elem*: $\llbracket c \in Q A; e \in \Sigma A; c' \in Q A; (c, e, c') \in D \rrbracket \Longrightarrow (c, [e], c') \in trclAD A D$ **by** *auto*

lemma *trclAD-uncons*: $(c, a \# w, c') \in trclAD A D \Longrightarrow \exists ch. (c, a, ch) \in D \wedge (ch, w, c') \in trclAD A D \wedge c \in Q A \wedge a \in \Sigma A$
by (*auto elim: trclAD.cases*)

lemma *trclAD-concat*: $\llbracket c. \llbracket (c, w1, c') \in trclAD A D; (c', w2, c'') \in trclAD A D \rrbracket \Longrightarrow (c, w1 @ w2, c'') \in trclAD A D$

proof (*induct w1*)

case *Nil* **thus** *?case* **by** (*subgoal-tac c=c'*) *auto*

next

case (*Cons a w*) **thus** *?case* **by** (*auto dest: trclAD-uncons*)

qed

lemma *trclAD-unconcat*: $!! c . (c, w1 @ w2, c') \in \text{trclAD } A \ D \implies \exists ch . (c, w1, ch) \in \text{trclAD } A \ D \wedge (ch, w2, c') \in \text{trclAD } A \ D$ **proof** (*induct w1*)
case *Nil* **hence** $(c, [], c) \in \text{trclAD } A \ D \wedge (c, w2, c') \in \text{trclAD } A \ D$ **by** (*auto dest: trclAD-elems*)
thus *?case* **by** *fast*
next
case (*Cons a w1*) **note** *IHP = this*
hence $(c, a \# (w1 @ w2), c') \in \text{trclAD } A \ D$ **by** *simp*
with *trclAD-uncons* **obtain** *chh* **where** $(c, a, chh) \in D \wedge (chh, w1 @ w2, c') \in \text{trclAD } A \ D \wedge c \in Q \ A \wedge a \in \Sigma \ A$ **by** *fast*
moreover with *IHP* **obtain** *ch* **where** $(chh, w1, ch) \in \text{trclAD } A \ D \wedge (ch, w2, c') \in \text{trclAD } A \ D$ **by** *fast*
ultimately have $(c, a \# w1, ch) \in \text{trclAD } A \ D \wedge (ch, w2, c') \in \text{trclAD } A \ D$ **by** *auto*
thus *?case* **by** *fast*
qed

lemma *trclAD-eq*: $\llbracket Q \ A = Q \ A'; \Sigma \ A = \Sigma \ A' \rrbracket \implies \text{trclAD } A \ D = \text{trclAD } A' \ D$ **by** (*unfold trclAD-def trclADp-def*) *simp*

lemma *trclAD-mono*: $D \subseteq D' \implies \text{trclAD } A \ D \subseteq \text{trclAD } A \ D'$
apply (*clarsimp*)
apply (*erule trclAD.induct*)
apply *auto*
done

lemma *trclAD-mono-adv*: $\llbracket D \subseteq D'; Q \ A = Q \ A'; \Sigma \ A = \Sigma \ A' \rrbracket \implies \text{trclAD } A \ D \subseteq \text{trclAD } A' \ D'$ **by** (*subgoal-tac trclAD A D = trclAD A' D*) (*auto dest: trclAD-eq trclAD-mono*)

4.3.1 Relation of *trclAD* and *trcl*

lemma *trclAD-by-trcl1*: $\text{trclAD } A \ D \subseteq (\text{trcl } (D \cap (Q \ A \times \Sigma \ A \times Q \ A))) \cap (Q \ A \times \text{lists } (\Sigma \ A) \times Q \ A)$
apply (*auto dest: trclAD-elems*)
apply (*erule trclAD.induct*)
apply (*auto simp add: trclAD-elems intro: trcl.cons*)
done

lemma *trclAD-by-trcl2*: $(\text{trcl } (D \cap (Q \ A \times \Sigma \ A \times Q \ A))) \cap (Q \ A \times \text{lists } (\Sigma \ A) \times Q \ A) \subseteq \text{trclAD } A \ D$ **proof** –
{ fix *c*
have $!! s \ s' . \llbracket (s, c, s') \in \text{trcl } (D \cap Q \ A \times \Sigma \ A \times Q \ A); s \in Q \ A; s' \in Q \ A; c \in \text{lists } (\Sigma \ A) \rrbracket \implies (s, c, s') \in \text{trclAD } A \ D$ **proof** (*induct c*)
case *Nil* **thus** *?case* **by** (*auto dest: trcl-empty-cons*)
next
case (*Cons e w*) **note** *IHP=this*
then obtain *sh* **where** *SPLIT*: $(s, e, sh) \in (D \cap Q \ A \times \Sigma \ A \times Q \ A) \wedge (sh, w, s') \in \text{trcl } (D \cap Q \ A \times \Sigma \ A \times Q \ A)$ **by** (*fast dest: trcl-uncons*)

hence $(sh, w, s') \in \text{trcl} (D \cap Q A \times \Sigma A \times Q A) \cap (Q A \times \text{lists} (\Sigma A) \times Q A)$ **by** $(\text{auto elim!}: \text{trcl-structE})$
hence $(sh, w, s') \in \text{trclAD} A D$ **by** $(\text{blast intro}: \text{IHP})$
with SPLIT show ?case by auto
qed
}
thus ?thesis by (auto)
qed

lemma trclAD-by-trcl: $\text{trclAD} A D = (\text{trcl} (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{lists} (\Sigma A) \times Q A))$
apply (rule equalityI)
apply (rule trclAD-by-trcl1)
apply (rule trclAD-by-trcl2)
done

lemma trclAD-by-trcl': $\text{trclAD} A D = (\text{trcl} (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{UNIV} \times \text{UNIV}))$
by (auto iff add: trclAD-by-trcl elim!: trcl-structE)

lemma trclAD-by-trcl'': $\llbracket D \subseteq Q A \times \Sigma A \times Q A \rrbracket \implies \text{trclAD} A D = \text{trcl} D \cap (Q A \times \text{UNIV} \times \text{UNIV})$
using trclAD-by-trcl'[of A D] by (simp add: Int-absorb2)

lemma trclAD-subset-trcl: $\text{trclAD} A D \subseteq \text{trcl} (D)$ **proof** –
have $\text{trclAD} A D \subseteq (\text{trcl} (D \cap (Q A \times \Sigma A \times Q A)))$ **by (auto simp add: trclAD-by-trcl)**
also with trcl-mono[of D \cap (Q A \times \Sigma A \times Q A) D] have ... \subseteq trcl D by auto
finally show ?thesis .
qed

4.4 Language of a FSM

constdefs

$\text{langs} A s == \{ w . (\exists f \in (F A) . (s, w, f) \in \text{trcl} A A) \}$
 $\text{lang} A == \text{langs} A (s0 A)$

lemma langs-alt-def: $(w \in \text{langs} A s) == (\exists f . f \in F A \ \& \ (s, w, f) \in \text{trcl} A A)$ **by (intro eq-reflection, unfold langs-def, auto)**

4.5 Example: Product automaton

constdefs

$\text{prod-fsm} A1 A2 == (\llbracket Q = Q A1 \times Q A2, \Sigma = \Sigma A1 \cap \Sigma A2, \delta = \{ ((s, t), a, (s', t')) . (s, a, s') \in \delta A1 \ \& \ (t, a, t') \in \delta A2 \}, s0 = (s0 A1, s0 A2), F = \{(s, t) . s \in F A1 \ \& \ t \in F A2 \} \rrbracket)$

lemma *prod-inter-1*: $!! s s' f f' . ((s,s'),w,(f,f')) \in \text{trclA } (\text{prod-fsm } A A') \implies (s,w,f) \in \text{trclA } A \wedge (s',w,f') \in \text{trclA } A' \text{ **proof** (induct w)}$
case *Nil* **note** $P=\text{this}$
moreover **hence** $s=f \wedge s'=f'$ **by** (*fast dest: trclAD-empty-cons*)
moreover **from** P **have** $s \in Q A \wedge s' \in Q A'$ **by** (*unfold prod-fsm-def, auto dest: trclAD-elems*)
ultimately **show** $?case$ **by** (*auto*)
next
case (*Cons e w*)
note $IHP=\text{this}$
then **obtain** $m m'$ **where** $I: ((s,s'),e,(m,m')) \in \delta (\text{prod-fsm } A A') \wedge (s,s') \in Q (\text{prod-fsm } A A') \wedge e \in \Sigma (\text{prod-fsm } A A') \wedge ((m,m'),w,(f,f')) \in \text{trclA } (\text{prod-fsm } A A')$ **by** (*fast dest: trclAD-uncons*)
hence $(s,e,m) \in \delta A \wedge (s',e,m') \in \delta A' \wedge s \in Q A \wedge s' \in Q A' \wedge e \in \Sigma A \wedge e \in \Sigma A'$ **by** (*unfold prod-fsm-def, simp*)
moreover **from** I IHP **have** $(m,w,f) \in \text{trclA } A \wedge (m',w,f') \in \text{trclA } A'$ **by** *auto*
ultimately **show** $?case$ **by** *auto*
qed

lemma *prod-inter-2*: $!! s s' f f' . (s,w,f) \in \text{trclA } A \wedge (s',w,f') \in \text{trclA } A' \implies ((s,s'),w,(f,f')) \in \text{trclA } (\text{prod-fsm } A A') \text{ **proof** (induct w)}$
case *Nil* **note** $P=\text{this}$
moreover **hence** $s=f \wedge s'=f'$ **by** (*fast dest: trclAD-empty-cons*)
moreover **from** P **have** $(s,s') \in Q (\text{prod-fsm } A A')$ **by** (*unfold prod-fsm-def, auto dest: trclAD-elems*)
ultimately **show** $?case$ **by** *simp*
next
case (*Cons e w*)
note $IHP=\text{this}$
then **obtain** $m m'$ **where** $I: (s,e,m) \in \delta A \wedge (m,w,f) \in \text{trclA } A \wedge (s',e,m') \in \delta A' \wedge (m',w,f') \in \text{trclA } A' \wedge s \in Q A \wedge s' \in Q A' \wedge e \in \Sigma A \wedge e \in \Sigma A'$ **by** (*fast dest: trclAD-uncons*)
hence $((s,s'),e,(m,m')) \in \delta (\text{prod-fsm } A A') \wedge (s,s') \in Q (\text{prod-fsm } A A') \wedge e \in \Sigma (\text{prod-fsm } A A')$ **by** (*unfold prod-fsm-def, simp*)
moreover **from** I IHP **have** $((m,m'),w,(f,f')) \in \text{trclA } (\text{prod-fsm } A A')$ **by** *auto*
ultimately **show** $?case$ **by** *auto*
qed

lemma *prod-F*: $(a,b) \in F (\text{prod-fsm } A B) = (a \in F A \wedge b \in F B)$ **by** (*unfold prod-fsm-def, auto*)
lemma *prod-FI*: $\llbracket a \in F A; b \in F B \rrbracket \implies (a,b) \in F (\text{prod-fsm } A B)$ **by** (*unfold prod-fsm-def, auto*)

lemma *prod-fsm-langs*: $\text{langs } (\text{prod-fsm } A B) (s,t) = \text{langs } A s \cap \text{langs } B t$
apply (*unfold langs-def*)
apply (*insert prod-inter-1 prod-F*)
apply (*fast intro: prod-inter-2 prod-FI*)
done

lemma *prod-FSM-intro*: $FSM\ A1 \implies FSM\ A2 \implies FSM\ (prod-fsm\ A1\ A2)$ by
(rule FSM.intro) (auto simp add: FSM-def prod-fsm-def)

end

5 Dynamic pushdown networks

theory *DPN*
imports *Main LTS SRS NDET FSM Misc*
begin

Dynamic pushdown networks (DPNs) are a model for parallel, context free processes where processes can create new processes.

They have been introduced in [1]. In this theory we formalize DPNs and the automata based algorithm for calculating a representation of the (regular) set of backward reachable configurations, starting at a regular set of configurations.

We describe the algorithm nondeterministically, and prove its termination and correctness.

5.1 Dynamic pushdown networks

5.1.1 Definition

record (c,l) *DPN-rec* =
csyms :: 'c set
ssyms :: 'c set
sep :: 'c
labels :: 'l set
rules :: ('c,'l) *SRS*

A dynamic pushdown network consists of a finite set of control symbols, a finite set of stack symbols, a separator symbol¹, a finite set of labels and a finite set of labelled string rewrite rules.

The set of control and stack symbols are disjoint, and both do not contain the separator. A string rewrite rule is either of the form $[p,\gamma] \hookrightarrow_a p1\#w1$ or $[p,\gamma] \hookrightarrow_a p1\#w1@\#\#p2\#w2$ where $p,p1,p2$ are control symbols, $w1,w2$ are sequences of stack symbols, a is a label and $\#$ is the separator.

locale *DPN* =
fixes *M*
fixes *separator* ($\#$)
defines *sep-def*: $\# == sep\ M$

¹In the final version of [1], no separator symbols are used. We use them here because we think it simplifies formalization of the proofs.

assumes *sym-finite*: *finite* (*csyms* *M*) *finite* (*ssyms* *M*)
assumes *sym-disjoint*: $csyms\ M \cap ssyms\ M = \{\#\}$
assumes *lab-finite*: *finite* (*labels* *M*)
assumes *rules-finite*: *finite* (*rules* *M*)
assumes *rule-fmt*: $r \in rules\ M \implies$
 $(\exists p\ \gamma\ a\ p'\ w. p \in csyms\ M \wedge \gamma \in ssyms\ M \wedge p' \in csyms\ M \wedge w \in lists\ (ssyms\ M)$
 $\wedge a \in labels\ M \wedge r = p\#\[\gamma] \hookrightarrow_a\ p'\#w)$
 $\vee (\exists p\ \gamma\ a\ p1\ w1\ p2\ w2. p \in csyms\ M \wedge \gamma \in ssyms\ M \wedge p1 \in csyms\ M \wedge w1 \in lists$
 $(ssyms\ M) \wedge p2 \in csyms\ M \wedge w2 \in lists\ (ssyms\ M) \wedge a \in labels\ M \wedge r = p\#\[\gamma] \hookrightarrow_a$
 $p1\#w1@#\#p2\#w2)$

lemma (in *DPN*) *sep-fold*: $sep\ M == \#\$ **by** (*simp* *add*: *sep-def*)

lemma (in *DPN*) *sym-disjoint'*: $sep\ M \notin csyms\ M \cup ssyms\ M$ **using** *sym-disjoint* **by** (*simp* *add*: *sep-def*)

5.1.2 Basic properties

lemma (in *DPN*) *syms-part*: $x \in csyms\ M \implies x \notin ssyms\ M$ $x \in ssyms\ M \implies x \notin csyms\ M$ **using** *sym-disjoint* **by** *auto*

lemma (in *DPN*) *syms-sep*: $\#\notin csyms\ M$ $\#\notin ssyms\ M$ **using** *sym-disjoint* **by** *auto*

lemma (in *DPN*) *syms-sep'*: $sep\ M \notin csyms\ M$ $sep\ M \notin ssyms\ M$ **using** *syms-sep* **by** (*auto* *simp* *add*: *sep-def*)

lemma (in *DPN*) *rule-cases*:

assumes *A*: $r \in rules\ M$
assumes *NOSPAWN*: $!!\ p\ \gamma\ a\ p'\ w. \llbracket p \in csyms\ M; \gamma \in ssyms\ M; p' \in csyms\ M;$
 $w \in lists\ (ssyms\ M); a \in labels\ M; r = p\#\[\gamma] \hookrightarrow_a\ p'\#w \rrbracket \implies P$
assumes *SPAWN*: $!!\ p\ \gamma\ a\ p1\ w1\ p2\ w2. \llbracket p \in csyms\ M; \gamma \in ssyms\ M; p1 \in csyms$
 $M; w1 \in lists\ (ssyms\ M); p2 \in csyms\ M; w2 \in lists\ (ssyms\ M); a \in labels\ M; r = p\#\[\gamma]$
 $\hookrightarrow_a\ p1\#w1@#\#p2\#w2 \rrbracket \implies P$
shows *P*
using *A* *NOSPAWN* *SPAWN*
by (*blast* *dest*!: *rule-fmt*)

lemma (in *DPN*) *rule-cases'*:

$\llbracket r \in rules\ M;$
 $!!\ p\ \gamma\ a\ p'\ w. \llbracket p \in csyms\ M; \gamma \in ssyms\ M; p' \in csyms\ M; w \in lists\ (ssyms\ M);$
 $a \in labels\ M; r = p\#\[\gamma] \hookrightarrow_a\ p'\#w \rrbracket \implies P;$
 $!!\ p\ \gamma\ a\ p1\ w1\ p2\ w2. \llbracket p \in csyms\ M; \gamma \in ssyms\ M; p1 \in csyms\ M; w1 \in lists\ (ssyms$
 $M); p2 \in csyms\ M; w2 \in lists\ (ssyms\ M); a \in labels\ M; r = p\#\[\gamma] \hookrightarrow_a\ p1\#w1@(\$
 $sep\ M)\#p2\#w2 \rrbracket \implies P \rrbracket$
 $\implies P$ **by** (*unfold* *sep-fold*) (*blast* *elim*!: *rule-cases*)

lemma (in *DPN*) *rule-prem-fmt*: $r \in rules\ M \implies \exists\ p\ \gamma\ a\ c'. p \in csyms\ M \wedge$
 $\gamma \in ssyms\ M \wedge a \in labels\ M \wedge set\ c' \subseteq csyms\ M \cup ssyms\ M \cup \{\#\} \wedge r = (p\#\[\gamma]$
 $\hookrightarrow_a\ c')$
apply (*erule* *rule-cases*)
by (*auto*)

lemma (in *DPN*) *rule-prem-fmt'*: $r \in \text{rules } M \implies \exists p \ \gamma \ a \ c'. \ p \in \text{csyms } M \wedge \gamma \in \text{ssyms } M \wedge a \in \text{labels } M \wedge \text{set } c' \subseteq \text{csyms } M \cup \text{ssyms } M \cup \{\text{sep } M\} \wedge r = (p\#\[\gamma] \hookrightarrow_a c')$ **by** (*unfold sep-fold, rule rule-prem-fmt*)

lemma (in *DPN*) *rule-prem-fmt2*: $[p,\gamma] \hookrightarrow_a c' \in \text{rules } M \implies p \in \text{csyms } M \wedge \gamma \in \text{ssyms } M \wedge a \in \text{labels } M \wedge \text{set } c' \subseteq \text{csyms } M \cup \text{ssyms } M \cup \{\#\}$ **by** (*fast dest: rule-prem-fmt*)

lemma (in *DPN*) *rule-prem-fmt2'*: $[p,\gamma] \hookrightarrow_a c' \in \text{rules } M \implies p \in \text{csyms } M \wedge \gamma \in \text{ssyms } M \wedge a \in \text{labels } M \wedge \text{set } c' \subseteq \text{csyms } M \cup \text{ssyms } M \cup \{\text{sep } M\}$ **by** (*unfold sep-fold, rule rule-prem-fmt2*)

lemma (in *DPN*) *rule-fmt-fs*: $[p,\gamma] \hookrightarrow_a p'\#c' \in \text{rules } M \implies p \in \text{csyms } M \wedge \gamma \in \text{ssyms } M \wedge a \in \text{labels } M \wedge p' \in \text{csyms } M \wedge \text{set } c' \subseteq \text{csyms } M \cup \text{ssyms } M \cup \{\#\}$

apply (*erule rule-cases*)

by (*auto*)

5.2 M-automata

We are interested in calculating the predecessor sets of regular sets of configurations. For this purpose, the regular sets of configurations are represented as finite state machines, that conform to certain constraints, depending on the underlying DPN. These FSMs are called M-automata.

5.2.1 Definition

record (*'s','c*) *MFSM-rec* = (*'s','c*) *FSM-rec* +
sstates :: *'s* set
cstates :: *'s* set
sp :: *'s* \Rightarrow *'c* \Rightarrow *'s*

M-automata are FSMs whose states are partitioned into control and stack states. For each control state s and control symbol p , there is a unique and distinguished stack state $sp \ A \ s \ p$, and a transition $(s,p,sp \ A \ s \ p) \in \delta$. The initial state is a control state, and the final states are all stack states. Moreover, the transitions are restricted: The only incoming transitions of control states are separator transitions from stack states. The only outgoing transitions are the $(s,p,sp \ A \ s \ p) \in \delta$ transitions mentioned above. The $sp \ A \ s \ p$ -states have no other incoming transitions.

locale *MFSM* = *DPN M* + *FSM A*
for *M A* +

assumes *alpha-cons*: $\Sigma \ A = \text{csyms } M \cup \text{ssyms } M \cup \{\#\}$

assumes *states-part*: $\text{sstates } A \cap \text{cstates } A = \{\}$ $Q \ A = \text{sstates } A \cup \text{cstates } A$

assumes *uniqueSp*: $\llbracket s \in \text{cstates } A; p \in \text{csyms } M \rrbracket \implies sp \ A \ s \ p \in \text{sstates } A \llbracket p \in \text{csyms } M; p' \in \text{csyms } M; s \in \text{cstates } A; s' \in \text{cstates } A; sp \ A \ s \ p = sp \ A \ s' \ p' \rrbracket \implies s = s' \wedge p = p'$

assumes *delta-fmt*: $\delta A \subseteq (sstates A \times ssyms M \times (sstates A - \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \cup (sstates A \times \{\#\} \times cstates A) \cup \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$
 $\delta A \supseteq \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$
assumes *s0-fmt*: $s0 A \in cstates A$
assumes *F-fmt*: $F A \subseteq sstates A$ — This deviates slightly from [1], as we cannot represent the empty configuration here. However, this restriction is harmless, since the only predecessor of the empty configuration is the empty configuration itself.
constrains *M*::('c,'l,'e1) *DPN-rec-scheme*
constrains *A*::('s,'c,'e2) *MFSM-rec-scheme*

lemma (in *MFSM*) *alpha-cons'*: $\Sigma A = csyms M \cup ssyms M \cup \{sep M\}$ **by** (*unfold sep-fold, rule alpha-cons*)
lemma (in *MFSM*) *delta-fmt'*: $\delta A \subseteq (sstates A \times ssyms M \times (sstates A - \{sp A s p \mid s p . s \in cstates A \wedge p \in csyms M\})) \cup (sstates A \times \{sep M\} \times cstates A) \cup \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$
 $\delta A \supseteq \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$ **by** (*unfold sep-fold, (rule delta-fmt)+*)

5.2.2 Basic properties

lemma (in *MFSM*) *finite-cs-states*: *finite* (*sstates A*) *finite* (*cstates A*)
proof —
have $sstates A \subseteq Q A \wedge cstates A \subseteq Q A$ **by** (*auto simp add: states-part*)
then show *finite* (*sstates A*) *finite* (*cstates A*) **by** (*auto dest: finite-subset intro: finite-states*)
qed

lemma (in *MFSM*) *sep-out-syms*: $x \in csyms M \implies x \neq \# \mid x \in ssyms M \implies x \neq \#$
by (*auto iff add: syms-sep*)
lemma (in *MFSM*) *sepI*: $\llbracket x \in \Sigma A; x \notin csyms M; x \notin ssyms M \rrbracket \implies x = \#$ **using** *alpha-cons* **by** *auto*
lemma (in *MFSM*) *sep-out-syms'*: $x \in csyms M \implies x \neq sep M \mid x \in ssyms M \implies x \neq sep M$ **by** (*unfold sep-fold, (fast dest: sep-out-syms) +*)
lemma (in *MFSM*) *sepI'*: $\llbracket x \in \Sigma A; x \notin csyms M; x \notin ssyms M \rrbracket \implies x = sep M$ **using** *alpha-cons'* **by** *auto*

lemma (in *MFSM*) *states-partI1*: $x \in sstates A \implies \neg x \in cstates A$ **using** *states-part* **by** (*auto*)
lemma (in *MFSM*) *states-partI2*: $x \in cstates A \implies \neg x \in sstates A$ **using** *states-part* **by** (*auto*)
lemma (in *MFSM*) *states-part-elim*[*elim*]: $\llbracket q \in Q A; q \in sstates A \implies P; q \in cstates A \implies P \rrbracket \implies P$ **using** *states-part* **by** (*auto*)

lemmas (in *MFSM*) *mfsm-cons* = *sep-out-syms sepI sep-out-syms' sepI' states-partI1 states-partI2 syms-part syms-sep uniqueSp*
lemmas (in *MFSM*) *mfsm-cons'* = *sep-out-syms sepI sep-out-syms' sepI' states-partI1 states-partI2 syms-part uniqueSp*

lemma (in *MFSM*) *delta-cases*: $\llbracket (q,p,q') \in \delta A; q \in \text{sstates } A \wedge p \in \text{ssyms } M \wedge q' \in \text{sstates } A \wedge q' \notin \{sp A s p \mid s p . s \in \text{cstates } A \wedge p \in \text{csyms } M\} \implies P;$
 $q \in \text{sstates } A \wedge p = \# \wedge q' \in \text{cstates } A \implies P;$
 $q \in \text{cstates } A \wedge p \in \text{csyms } M \wedge q' = sp A q p \implies$

$P \rrbracket \implies P$

using *delta-fmt by auto*

lemma (in *MFSM*) *delta-elems*: $(q,p,q') \in \delta A \implies q \in \text{sstates } A \wedge ((p \in \text{ssyms } M \wedge q' \in \text{sstates } A \wedge (q' \notin \{sp A s p \mid s p . s \in \text{cstates } A \wedge p \in \text{csyms } M\})) \vee (p = \# \wedge q' \in \text{cstates } A)) \vee (q \in \text{cstates } A \wedge p \in \text{csyms } M \wedge q' = sp A q p)$
using *delta-fmt by auto*

lemma (in *MFSM*) *delta-cases'*: $\llbracket (q,p,q') \in \delta A; q \in \text{sstates } A \wedge p \in \text{ssyms } M \wedge q' \in \text{sstates } A \wedge q' \notin \{sp A s p \mid s p . s \in \text{cstates } A \wedge p \in \text{csyms } M\} \implies P;$
 $q \in \text{sstates } A \wedge p = \text{sep } M \wedge q' \in \text{cstates } A \implies P;$
 $q \in \text{cstates } A \wedge p \in \text{csyms } M \wedge q' = sp A q p \implies$

$P \rrbracket \implies P$

using *delta-fmt' by auto*

lemma (in *MFSM*) *delta-elems'*: $(q,p,q') \in \delta A \implies q \in \text{sstates } A \wedge ((p \in \text{ssyms } M \wedge q' \in \text{sstates } A \wedge (q' \notin \{sp A s p \mid s p . s \in \text{cstates } A \wedge p \in \text{csyms } M\})) \vee (p = \text{sep } M \wedge q' \in \text{cstates } A)) \vee (q \in \text{cstates } A \wedge p \in \text{csyms } M \wedge q' = sp A q p)$
using *delta-fmt' by auto*

5.2.3 Some implications of the M-automata conditions

This list of properties is taken almost literally from [1].

Each control state s has $sp A s p$ as its unique p -successor

lemma (in *MFSM*) *cstate-succ-ex*: $\llbracket p \in \text{csyms } M; s \in \text{cstates } A \rrbracket \implies (s,p,sp A s p) \in \delta A$

using *delta-fmt by (auto)*

lemma (in *MFSM*) *cstate-succ-ex'*: $\llbracket p \in \text{csyms } M; s \in \text{cstates } A; \delta A \subseteq D \rrbracket \implies (s,p,sp A s p) \in D$ **using** *cstate-succ-ex by auto*

lemma (in *MFSM*) *cstate-succ-unique*: $\llbracket s \in \text{cstates } A; (s,p,x) \in \delta A \rrbracket \implies p \in \text{csyms } M \wedge x = sp A s p$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

Transitions labeled with control symbols only leave from control states

lemma (in *MFSM*) *csym-from-cstate*: $\llbracket (s,p,s') \in \delta A; p \in \text{csyms } M \rrbracket \implies s \in \text{cstates } A$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

s is the only predecessor of $sp A s p$

lemma (in *MFSM*) *sp-pred-ex*: $\llbracket s \in \text{cstates } A; p \in \text{csyms } M \rrbracket \implies (s,p,sp A s p) \in \delta A$ **using** *delta-fmt by auto*

lemma (in *MFSM*) *sp-pred-unique*: $\llbracket s \in cstates\ A; p \in csyms\ M; (s', p', sp\ A\ s\ p) \in \delta\ A \rrbracket \implies s' = s \wedge p' = p \wedge s' \in cstates\ A \wedge p' \in csyms\ M$ **by** (*erule delta-cases*) (*auto dest: mfsm-cons'*)

Only separators lead from stack states to control states

lemma (in *MFSM*) *sep-in-between*: $\llbracket s \in sstates\ A; s' \in cstates\ A; (s, p, s') \in \delta\ A \rrbracket \implies p = \#$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

lemma (in *MFSM*) *sep-to-cstate*: $\llbracket (s, \#, s') \in \delta\ A \rrbracket \implies s \in sstates\ A \wedge s' \in cstates\ A$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

Stack states do not have successors labelled with control symbols

lemma (in *MFSM*) *sstate-succ*: $\llbracket s \in sstates\ A; (s, \gamma, s') \in \delta\ A \rrbracket \implies \gamma \notin csyms\ M$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

lemma (in *MFSM*) *sstate-succ2*: $\llbracket s \in sstates\ A; (s, \gamma, s') \in \delta\ A; \gamma \neq \# \rrbracket \implies \gamma \in ssyms\ M \wedge s' \in sstates\ A$ **by** (*auto elim: delta-cases dest: mfsm-cons'*)

M-automata do not accept the empty word

lemma (in *MFSM*) *not-empty[iff]*: $\llbracket \notin lang\ A \rrbracket$

apply (*unfold lang-def langs-def*)

apply (*clarsimp*)

apply (*insert s0-fmt F-fmt*)

apply (*subgoal-tac s0 A = f*)

apply (*auto dest: mfsm-cons'*)

done

The paths through an M-automata have a very special form: Paths starting at a stack state are either labelled entirely with stack symbols, or have a prefix labelled with stack symbols followed by a separator

lemma (in *MFSM*) *path-from-sstate*: $\llbracket !s . \llbracket s \in sstates\ A; (s, w, f) \in trclA\ A \rrbracket \implies (f \in sstates\ A \wedge w \in lists\ (ssyms\ M)) \vee (\exists w1\ w2\ t. w = w1 @ \# \# w2 \wedge w1 \in lists\ (ssyms\ M) \wedge t \in sstates\ A \wedge (s, w1, t) \in trclA\ A \wedge (t, \# \# w2, f) \in trclA\ A) \rrbracket$

proof (*induct w*)

case Nil thus *?case by* (*subgoal-tac s=f*) *auto*

next

case (*Cons e w*)

note *IHP[rule-format]=this*

then obtain *s'* **where** *STEP*: $(s, e, s') \in (\delta\ A) \wedge s \in Q\ A \wedge e \in \Sigma\ A \wedge (s', w, f) \in trclA\ A$ **by** (*fast dest: trclAD-uncons*)

show *?case proof* (*cases e=#*)

assume $e = \#$

with IHP have $e \# w = \# @ \# \# w \wedge \llbracket \in lists\ (ssyms\ M) \wedge s \in sstates\ A \wedge (s, \#, s) \in trclA\ A \wedge (s, e \# w, f) \in trclA\ A$ **using** *states-part by* (*auto*)

thus *?case by force*

next

assume $e \neq \#$

with IHP STEP sstate-succ2 have *EC*: $e \in ssyms\ M \wedge s' \in sstates\ A$ **by** *blast*

with IHP STEP have $(f \in sstates\ A \wedge w \in lists\ (ssyms\ M)) \vee (\exists w1\ w2\ t. w = w1 @ \# \# w2 \wedge w1 \in lists\ (ssyms\ M) \wedge t \in sstates\ A \wedge (s', w1, t) \in trclA\ A \wedge (t, \# \# w2, f) \in trclA\ A)$ **(is** *?C1* \vee *?C2*) **by** *auto*

```

moreover {
  assume ?C1
  with EC have  $f \in \text{sstates } A \wedge e \# w \in \text{lists } (\text{ssyms } M)$  by auto
} moreover {
  assume ?C2
  then obtain  $w1\ w2\ t$  where CASE:  $w = w1 @ \# \# w2 \wedge w1 \in \text{lists } (\text{ssyms } M) \wedge t \in \text{sstates } A \wedge (s', w1, t) \in \text{trclA } A \wedge (t, \# \# w2, f) \in \text{trclA } A$  by (fast)
  with EC have  $e \# w = (e \# w1) @ \# \# w2 \wedge e \# w1 \in \text{lists } (\text{ssyms } M)$  by auto
  moreover from CASE STEP IHP have  $(s, e \# w1, t) \in \text{trclA } A$  using states-part by auto
  moreover note CASE
  ultimately have  $\exists w1\ w2\ t. e \# w = w1 @ \# \# w2 \wedge w1 \in \text{lists } (\text{ssyms } M) \wedge t \in \text{sstates } A \wedge (s, w1, t) \in \text{trclA } A \wedge (t, \# \# w2, f) \in \text{trclA } A$  by fast
} ultimately show ?case by blast
qed
qed

```

Using *MFSM.path-from-sstate*, we can describe the format of paths from control states, too. A path from a control state s to some final state starts with a transition $(s, p, sp\ A\ s\ p)$ for some control symbol p . It then continues with a sequence of transitions labelled by stack symbols. It then either ends or continues with a separator transition, bringing it to a control state again, and some further transitions from there on.

lemma (in *MFSM*) *path-from-cstate*:

```

assumes A:  $s \in \text{cstates } A \wedge (s, c, f) \in \text{trclA } A \wedge f \in \text{sstates } A$ 
assumes SINGLE:  $!! p\ w. \llbracket c = p \# w; p \in \text{csyms } M; w \in \text{lists } (\text{ssyms } M); (s, p, sp\ A\ s\ p) \in \delta\ A; (sp\ A\ s\ p, w, f) \in \text{trclA } A \rrbracket \implies P$ 
assumes CONC:  $!! p\ w\ cr\ t\ s'. \llbracket c = p \# w @ \# \# cr; p \in \text{csyms } M; w \in \text{lists } (\text{ssyms } M); t \in \text{sstates } A; s' \in \text{cstates } A; (s, p, sp\ A\ s\ p) \in \delta\ A; (sp\ A\ s\ p, w, t) \in \text{trclA } A; (t, \# \# s') \in \delta\ A; (s', cr, f) \in \text{trclA } A \rrbracket \implies P$ 
shows P

```

proof (*cases c*)

case *Nil* **thus** *P* **using** *A* **by** (*subgoal-tac s=f, auto dest: m fsm-cons'*)

next

```

case (Cons p w) note CFMT=this
with cstate-succ-unique A have SPLIT:  $p \in \text{csyms } M \wedge (s, p, sp\ A\ s\ p) \in \delta\ A \wedge (sp\ A\ s\ p, w, f) \in \text{trclA } A$  by (blast dest: trclAD-uncons)
with path-from-sstate A CFMT uniqueSp have CASES:  $(f \in \text{sstates } A \wedge w \in \text{lists } (\text{ssyms } M)) \vee (\exists w1\ w2\ t. w = w1 @ \# \# w2 \wedge w1 \in \text{lists } (\text{ssyms } M) \wedge t \in \text{sstates } A \wedge (sp\ A\ s\ p, w1, t) \in \text{trclA } A \wedge (t, \# \# w2, f) \in \text{trclA } A)$  (is ?C1  $\vee$  ?C2) by blast

```

```

moreover {
  assume CASE: ?C1
  with SPLIT SINGLE A CFMT have P by fast
} moreover {
  assume CASE: ?C2
  then obtain  $w1\ w2\ t$  where WFMT:  $w = w1 @ \# \# w2 \wedge w1 \in \text{lists } (\text{ssyms } M) \wedge t \in \text{sstates } A \wedge (sp\ A\ s\ p, w1, t) \in \text{trclA } A \wedge (t, \# \# w2, f) \in \text{trclA } A$  by fast
  with sep-to-cstate obtain  $s'$  where  $s' \in \text{cstates } A \wedge (t, \# \# s') \in \delta\ A \wedge (s', w2, f) \in \text{trclA } A$ 

```

A by (*fast dest: trclAD-uncons*)
with SPLIT CASE WFMT have $p\#w=p\#w1@\#\#w2 \wedge p \in csyms\ M \wedge w1 \in lists$
 $(ssyms\ M) \wedge t \in sstates\ A \wedge s' \in cstates\ A \wedge (s,p,sp\ A\ s\ p) \in \delta\ A \wedge (sp\ A\ s\ p,w1,t) \in trclA$
 $A \wedge (t,\#,s') \in \delta\ A \wedge (s',w2,f) \in trclA\ A$ **by auto**
with CFMT CONC have P **by** (*fast*)
} ultimately show P **by** (*blast*)
qed

5.3 pre^* -sets of regular sets of configurations

Given a regular set L of configurations and a set Δ of string rewrite rules, $pre^* \Delta L$ is the set of configurations that can be rewritten to some configuration in L , using rules from Δ arbitrarily often.

We first define this set inductively based on rewrite steps, and then provide the characterization described above as a lemma.

inductive-set $pre\text{-}star :: ('c,'l)\ SRS \Rightarrow ('s,'c,'e)\ FSM\text{-}rec\text{-}scheme \Rightarrow 'c\ list\ set$
 (pre^*)
for $\Delta\ L$
where
 $pre\text{-}refl: c \in lang\ L \Longrightarrow c \in pre^* \Delta L \mid$
 $pre\text{-}step: \llbracket c' \in pre^* \Delta L; (c,a,c') \in tr\ \Delta \rrbracket \Longrightarrow c \in pre^* \Delta L$

Alternative characterization of $pre^* \Delta L$

lemma $pre\text{-}star\text{-}alt: pre^* \Delta L == \{c . \exists c' \in lang\ L . \exists as . (c,as,c') \in trcl\ (tr\ \Delta)\}$
apply (*rule eq-reflection*)
apply (*auto elim!: pre-star.induct intro: trcl.intros*)
apply (*force intro: trcl.intros*)

proof –

fix $x\ c'$ *as*
show $\llbracket x \hookrightarrow_{as}\ c' \in trcl\ (tr\ \Delta); c' \in lang\ L \rrbracket \Longrightarrow x \in pre^* \Delta L$
by (*induct rule: trcl.induct*) (*auto intro: pre-step pre-refl*)
qed

lemma $pre\text{-}star\text{-}altI: \llbracket c' \in lang\ L; c \hookrightarrow_{as}\ c' \in trcl\ (tr\ \Delta) \rrbracket \Longrightarrow c \in pre^* \Delta L$ **by** (*unfold pre-star-alt, auto*)

lemma $pre\text{-}star\text{-}altE: \llbracket c \in pre^* \Delta L; !!c' as. \llbracket c' \in lang\ L; c \hookrightarrow_{as}\ c' \in trcl\ (tr\ \Delta) \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$ **by** (*unfold pre-star-alt, auto*)

5.4 Nondeterministic algorithm for pre^*

In this section, we formalize the saturation algorithm for computing $pre^* \Delta L$ from [1]. Roughly, the algorithm works as follows:

1. Set $D = \delta A$
2. Choose a rule $([p, \gamma], a, c') \in rules\ M$ and states $q, q' \in Q\ A$, such that D can read the configuration c' from state q and end in state q' (i.e.

$(q, c', q') \in \text{trclAD } A D$) and such that $(sp A q p, \gamma, q') \notin D$. If this is not possible, terminate.

3. Add the transition $(sp A q p, \gamma, q') \notin D$ to D and continue with step 2

Intuitively, the behaviour of this algorithm can be explained as follows: If there is a configuration $c_1 @ c' @ c_2 \in \text{pre}^* \Delta L$, and a rule $(p \# \gamma, a, c') \in \Delta$, then we also have $c_1 @ p \# \gamma @ c_2 \in \text{pre}^* \Delta L$. The effect of step 3 is exactly adding these configurations $c_1 @ p \# \gamma @ c_2$ to the regular set of configurations.

We describe the algorithm nondeterministically by its step relation $ps\text{-}R$. Each step describes the addition of one transition.

In this approach, we directly restrict the domain of the step-relation to transition relations below some upper bound $ps\text{-}upper$, that we will define later. We will later show, that the initial transition relation of an M-automata is below this upper bound, and that the step-relation preserves the property of being below this upper bound.

consts

$ps\text{-}upper :: ('c, 'l, 'e1) \text{DPN-rec-scheme} \Rightarrow ('s, 'c, 'e2) \text{MFSM-rec-scheme} \Rightarrow ('s, 'c) \text{LTS}$

inductive-set $ps\text{-}R :: ('c, 'l, 'e1) \text{DPN-rec-scheme} \Rightarrow ('s, 'c, 'e2) \text{MFSM-rec-scheme} \Rightarrow (('s, 'c) \text{LTS} * ('s, 'c) \text{LTS}) \text{set for } M A$

where

$\llbracket [p, \gamma] \hookrightarrow_a c' \in \text{rules } M; (q, c', q') \in \text{trclAD } A D; (sp A q p, \gamma, q') \notin D; D \subseteq ps\text{-}upper M A \rrbracket \Longrightarrow (D, \text{insert } (sp A q p, \gamma, q') D) \in ps\text{-}R M A$

lemma $ps\text{-}R\text{-dom-below}: (D, D') \in ps\text{-}R M A \Longrightarrow D \subseteq ps\text{-}upper M A$ **by** (*auto elim: ps-R.cases*)

5.4.1 Termination

Termination of our algorithm is equivalent to well-foundedness of its (converse) step relation, that is, we have to show $wf ((ps\text{-}R M A)^{-1})$.

We define $ps\text{-}upper M A$ as a finite set, and show that the initial transition relation δA of an M-automata is below $ps\text{-}upper M A$, and that $ps\text{-}R M A$ preserves the property of being below the finite set $ps\text{-}upper M A$.

In the following, we also establish some properties of transition relations below $ps\text{-}upper M A$, that will be used later in the correctness proof.

Note that we use the more fine-grained $ps\text{-}upper M A$ as upper bound for the termination proof rather than $Q A \times \Sigma A \times Q A$, as $sp A q p$ is only specified for control states q and control symbols p . Hence we need the finer structure of $ps\text{-}upper M A$ to guarantee that sp is only applied to arguments

it is specified for. Anyway, the fine-grained $ps\text{-upper } M A$ bound is also needed for the correctness proof.

defs

$ps\text{-upper-def}$: $ps\text{-upper } M A == (sstates A \times ssyms M \times sstates A) \cup (sstates A \times \{sep M\} \times cstates A) \cup \{(s,p,sp A s p) \mid s p . s \in cstates A \wedge p \in csyms M\}$

lemma (in MFSM) $ps\text{-upper-cases}$: $\llbracket (s,e,s') \in ps\text{-upper } M A;$
 $\llbracket s \in sstates A; e \in ssyms M; s' \in sstates A \rrbracket \implies P;$
 $\llbracket s \in sstates A; e = \#; s' \in cstates A \rrbracket \implies P;$
 $\llbracket s \in cstates A; e \in csyms M; s' = sp A s e \rrbracket \implies P$
 $\rrbracket \implies P$
by ($unfold ps\text{-upper-def sep-def, auto}$)

lemma (in MFSM) $ps\text{-upper-cases}'$: $\llbracket (s,e,s') \in ps\text{-upper } M A;$
 $\llbracket s \in sstates A; e \in ssyms M; s' \in sstates A \rrbracket \implies P;$
 $\llbracket s \in sstates A; e = sep M; s' \in cstates A \rrbracket \implies P;$
 $\llbracket s \in cstates A; e \in csyms M; s' = sp A s e \rrbracket \implies P$
 $\rrbracket \implies P$
apply ($rule ps\text{-upper-cases}$)
by ($unfold sep-def auto$)

lemma (in MFSM) $ps\text{-upper-below-trivial}$: $ps\text{-upper } M A \subseteq Q A \times \Sigma A \times Q A$
by ($unfold ps\text{-upper-def, auto simp add: states-part alpha-cons uniqueSp sep-def}$)

lemma (in MFSM) $ps\text{-upper-finite}$: $finite (ps\text{-upper } M A)$ **using** $ps\text{-upper-below-trivial}$
 $finite\text{-delta-dom}$ **by** ($auto simp add: finite-subset$)

The initial transition relation of the M-automaton is below $ps\text{-upper } M A$

lemma (in MFSM) $initial\text{-delta-below}$: $\delta A \subseteq ps\text{-upper } M A$ **using** $delta\text{-fmt}$ **by**
($unfold ps\text{-upper-def sep-def auto}$)

Some lemmas about structure of transition relations below $ps\text{-upper } M A$

lemma (in MFSM) $cstate\text{-succ-unique}'$: $\llbracket s \in cstates A; (s,p,x) \in D; D \subseteq ps\text{-upper } M A \rrbracket \implies p \in csyms M \wedge x = sp A s p$ **by** ($auto elim: ps\text{-upper-cases dest: mfsm-cons}'$)
lemma (in MFSM) $csym\text{-from-cstate}'$: $\llbracket (s,p,s') \in D; D \subseteq ps\text{-upper } M A; p \in csyms M \rrbracket \implies s \in cstates A$ **by** ($auto elim: ps\text{-upper-cases dest: mfsm-cons}'$)

The only way to end up in a control state is after executing a separator.

lemma (in MFSM) $ctrl\text{-after-sep}$: **assumes** $BELOW: D \subseteq ps\text{-upper } M A$

assumes $A: (q,c',q') \in trclAD A D \quad c' \neq []$

shows $q' \in cstates A = (last c' = \#)$

proof –

from A **have** $(q, butlast c' @ [last c'], q') \in trclAD A D$ **by** $auto$

with A **obtain** qh **where** $(qh, [last c'], q') \in trclAD A D$ **by** ($blast dest: trclAD-unconcat$)

hence $(qh, last c', q') \in D$ **by** ($fast dest: trclAD-single$)

with $BELOW$ **have** $IS: (qh, last c', q') \in ps\text{-upper } M A$ **by** $fast$

thus $?thesis$ **by** ($erule-tac ps\text{-upper-cases}$) ($auto dest: mfsm-cons' simp add: sep-out-syms$)

qed

When applying a rules right hand side to a control state, we will get to a stack state

lemma (in *MFSM*) *ctrl-rule*: **assumes** *BELOW*: $D \subseteq ps\text{-upper } M A$
assumes *A*: $([p, \gamma], a, c') \in rules\ M\ q \in cstates\ A\ (q, c', q') \in trclAD\ A\ D$
shows $q' \in sstates\ A$

proof –

show *?thesis* **proof** (*insert A, erule rule-cases, auto*)

fix $p' w$

assume *C*: $q \hookrightarrow_{p'} \# w\ q' \in trclAD\ A\ D\ \forall x \in set\ w. x \in ssyms\ M\ p' \in csyms$

M

hence $last\ (p' \# w) \neq \# \wedge q' \in Q\ A$ **by** (*unfold sep-def*) (*auto dest: mfsm-cons' trclAD-elems*)

with *C* *BELOW* *ctrl-after-sep*[*of D q p' # w q'*] **show** ($q' \in sstates\ A$) **by** (*fast dest: mfsm-cons'*)

next

fix $p1\ w1\ p2\ w2$

assume *C*: $q \hookrightarrow_{p1} \# w1\ @ \# \# p2 \# w2\ q' \in trclAD\ A\ D\ \forall x \in set\ w2. x \in ssyms\ M\ p2 \in csyms\ M$

hence $last\ (p1 \# w1\ @ \# \# p2 \# w2) \neq sep\ M \wedge q' \in Q\ A$ **by** (*auto dest: mfsm-cons' trclAD-elems*)

with *C* *BELOW* *ctrl-after-sep*[*of D q p1 # w1 @ # # p2 # w2 q'*] **show** ($q' \in sstates\ A$) **by** (*unfold sep-def, fast dest: mfsm-cons'*)

qed

qed

ps-R *M* *A* preserves the property of being below *ps-upper* *M* *A*, and the transition relation becomes strictly greater in each step

lemma (in *MFSM*) *ps-R-below*: **assumes** *E*: $(D, D') \in ps\text{-R } M A$
shows $D \subset D' \wedge D' \subseteq ps\text{-upper } M A$

proof –

from *E* **have** *BELOW*: $D \subseteq ps\text{-upper } M A$ **by** (*simp add: ps-R-dom-below*)

show *?thesis* **using** *E* *BELOW* **proof** (*auto elim!: ps-R.cases*)

fix $p\ \gamma\ a\ c'\ q\ q'$

assume *A*: $[p, \gamma] \hookrightarrow_a\ c' \in rules\ M\ q \hookrightarrow_{c'}\ q' \in trclAD\ A\ D$

obtain $p'\ cr'$ **where** *CSPLIT*: $p \in csyms\ M \wedge p' \in csyms\ M \wedge c' = p' \# cr' \wedge \gamma \in ssyms\ M$ **by** (*insert A*) (*erule rule-cases, fast+*)

with *BELOW* *A* **obtain** qh **where** *SPLIT*: $(q, p', qh) \in D\ (q, p', qh) \in ps\text{-upper } M A$ **by** (*fast dest: trclAD-uncons*)

with *CSPLIT* **have** *QC*: $q \in cstates\ A \wedge qh = sp\ A\ q\ p'$ **by** (*auto elim: ps-upper-cases dest: syms-part iff add: syms-sep*)

with *BELOW* *A* *ctrl-rule*[*of D p \gamma a c' q q'*] **have** *Q'S*: $q' \in sstates\ A$ **by** *simp*

from *QC* *CSPLIT* **have** $sp\ A\ q\ p \in sstates\ A$ **by** (*simp add: uniqueSp*)

with *Q'S* *CSPLIT* **show** $sp\ A\ q\ p \hookrightarrow_{\gamma}\ q' \in ps\text{-upper } M A$ **by** (*unfold ps-upper-def, simp*)

qed

qed

As a result of this section, we get the well-foundedness of $ps\text{-}R\ M\ A$, and that the transition relations that occur during the saturation algorithm stay above the initial transition relation $\delta\ A$ and below $ps\text{-}upper\ M\ A$

theorem (in *MFSM*) $ps\text{-}R\text{-}wf: wf\ ((ps\text{-}R\ M\ A)^{-1})$ **using** $ps\text{-}upper\text{-}finite\ sat\text{-}wf$ [**where** $\alpha=id, simplified$] $ps\text{-}R\text{-}below$ **by** (*blast*)

theorem (in *MFSM*) $ps\text{-}R\text{-}above\text{-}inv: is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ (\lambda D. \delta\ A \subseteq D)$ **by** (*auto intro: invI elim: ps-R.cases*)

theorem (in *MFSM*) $ps\text{-}R\text{-}below\text{-}inv: is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ (\lambda D. D \subseteq ps\text{-}upper\ M\ A)$ **by** (*rule invI*) (*auto simp add: initial-delta-below ps-R-below*)

We can also show that the algorithm is defined for every possible initial automata

theorem (in *MFSM*) $total: \exists D. (\delta\ A, D) \in ndet\text{-}algo(ps\text{-}R\ M\ A)$ **using** $ps\text{-}R\text{-}wf$ $ndet\text{-}algo\text{-}total$ **by** (*blast*)

5.4.2 Soundness

The soundness (over-approximation) proof works by induction over the definition of pre^* .

In the reflexive case, a configuration from the original language is also in the saturated language, because no transitions are killed during saturation.

In the step case, we assume that a configuration c' is in the saturated language, and show for a rewriting step $c \xrightarrow{a} c'$ that also c is in the saturated language.

theorem (in *MFSM*) $sound: \llbracket c \in pre\text{-}star\ (rules\ M)\ A; (\delta\ A, s') \in ndet\text{-}algo\ (ps\text{-}R\ M\ A) \rrbracket \implies c \in lang\ (A\ (\delta := s'))$

proof –

let $?A' = A\ (\delta := s')$

assume $A: (\delta\ A, s') \in ndet\text{-}algo\ (ps\text{-}R\ M\ A)$

 — Some little helpers

from $A\ ps\text{-}R\text{-}above\text{-}inv$ **have** $SUBSET: \delta\ A \subseteq s'$ **by** (*unfold ndet-algo-def*) (*auto dest: inv*)

have $TREQ: !!D. trclAD\ A\ D = trclAD\ ?A'\ D$ **by** (*rule trclAD-eq, simp-all*)

from $A\ ps\text{-}R\text{-}below\text{-}inv$ **have** $SATSETU: \delta\ ?A' \subseteq ps\text{-}upper\ M\ A$ **by** (*erule-tac ndet-algoE*) (*auto dest: inv iff add: initial-delta-below*)

assume $c \in pre\text{-}star\ (rules\ M)\ A$

 — Make an induction over the definition of pre^*

thus $?thesis$ **proof** (*induct c rule: pre-star.induct*)

fix c **assume** $c \in lang\ A$ — Reflexive case: The configuration comes from the original regular language

then obtain f **where** $F: f \in F\ A \wedge (s0\ A, c, f) \in trclA\ A$ **by** (*unfold lang-def langs-def, fast*) — That is, c can bring the initial automata from its start state to some final state f

with *SUBSET trclAD-mono-adv*[of $\delta A s' A ?A'$] **have** $(s0 A, c, f) \in trclA ?A'$ **by** (*auto*) — Because the original transition relation δA is a subset of the saturated one s' (*SUBSET*) and the transitive closure is monotonous, $(s0 A, c, f)$ is also in the transitive closure of the saturated transition relation

with *F show* $c \in lang ?A'$ **by** (*unfold lang-def langs-def*) *auto* — and thus in the language of the saturated automaton

next

— Step case:

fix $a c c'$

assume *IHP*: $c' \in pre^* (rules M) A$ $(c, a, c') \in tr (rules M)$ — We take some configurations c and $c' \in pre^* (rules M) A$ and assume that c can be rewritten to c' in one step

$c' \in lang ?A'$ — We further assume that c' is in the saturated language, and we have to show that also c is in that language

from *IHP obtain* f **where** *F*: $f \in F ?A' \wedge (s0 ?A', c', f) \in trclA ?A'$ **by** (*unfold lang-def langs-def, fast*) — Unfolding the definition of *lang*

from *IHP obtain* $w1 w2 r r'$ **where** *CREW*: $c = w1 @ (r @ w2) \wedge c' = w1 @ (r' @ w2) \wedge (r, a, r') \in rules M$ **by** (*auto elim!*: *tr.cases*) — Get the rewrite rule that rewrites c to c'

then obtain $p \gamma p' w'$ **where** *RFMT*: $p \in csyms M \wedge p' \in csyms M \wedge \gamma \in ssyms M \wedge r = [p, \gamma] \wedge r' = p' \# w'$ **by** (*auto elim!*: *rule-cases*) — This rewrite rule rewrites some control symbol p followed by a stack symbol γ to another control symbol p' and a sequence of further symbols w'

with *F CREW obtain* $q qh q'$ **where** *SPLIT*: $(s0 ?A', w1, q) \in trclA ?A' \wedge (q, p' \# w', q') \in trclA ?A' \wedge (q', w2, f) \in trclA ?A' \wedge (q, p', qh) \in \delta ?A'$

by (*blast dest: trclAD-unconcat trclAD-uncons*) — Get the states in the transition relation generated by the algorithm, that correspond to the splitting of c' as established in *CREW*

have *SHORTCUT*: $(q, [p, \gamma], q') \in trclA ?A'$ — In the transition relation generated by our algorithm, we can get from q to q' also by $[p, \gamma]$

proof —

have *S1*: $(q, p, sp A q p) \in \delta ?A'$ **and** *QINC*: $q \in cstates A$ — The first transition, from q with p to $sp A q p$ is already contained in the initial M-automata. We also need to know for further proofs, that q is a control state.

proof —

from *SPLIT SATSETU have* $(q, p', qh) \in ps-upper M A$ **by** *auto*

with *RFMT show* $q \in cstates A$ **by** (*auto elim!*: *ps-upper-cases dest: mfsm-cons' simp add: sep-def*)

with *RFMT have* $(q, p, sp A q p) \in \delta A$ **by** (*fast intro: cstate-succ-ex*)

with *SUBSET show* $(q, p, sp A q p) \in \delta ?A'$ **by** *auto*

qed

moreover

have *S2*: $(sp A q p, \gamma, q') \in \delta ?A'$ — The second transition, from $sp A q p$ with γ to q' has been added during the algorithm's execution

proof —

from *A have* $s' \notin Domain (ps-R M A)$ **by** (*blast dest: termstate-ndet-algo*)

moreover from *CREW RFMT SPLIT TREQ SATSETU* **have** $(sp\ A\ q\ p, \gamma, q') \notin s' \implies (s', \text{insert}(sp\ A\ q\ p, \gamma, q')\ s') \in (ps\text{-}R\ M\ A)$ **by** (*auto intro: ps-R.intros*)

ultimately show *?thesis* **by** *auto*

qed

moreover

have $sp\ A\ q\ p \in Q\ ?A' \wedge q' \in Q\ ?A' \wedge q \in Q\ ?A' \wedge p \in \Sigma\ ?A' \wedge \gamma \in \Sigma\ ?A'$ —

The intermediate states and labels have also the correct types

proof —

from *S2 SATSETU* **have** $(sp\ A\ q\ p, \gamma, q') \in ps\text{-}upper\ M\ A$ **by** *auto*

with *QINC RFMT* **show** *?thesis* **by** (*auto elim: ps-upper-cases dest: mfsm-cons' simp add: states-part alpha-cons*)

qed

ultimately show *?thesis* **by** *simp*

qed

have $(s0\ ?A', w1 @ ([p, \gamma]) @ w2, f) \in trclA\ ?A'$ — Now we put the pieces together and construct a path from $s0\ A$ with $w1$ to q , from there with $[p, \gamma]$ to q' and then with $w2$ to the final state f

proof —

from *SHORTCUT SPLIT* **have** $(q, ([p, \gamma]) @ w2, f) \in trclA\ ?A'$ **by** (*fast dest: trclAD-concat*)

with *SPLIT* **show** *?thesis* **by** (*fast dest: trclAD-concat*)

qed

with *CREW RFMT* **have** $(s0\ ?A', c, f) \in trclA\ ?A'$ **by** *auto* — this is because $c = w1 @ [p, \gamma] @ w2$

with *F* **show** $c \in lang\ ?A'$ **by** (*unfold lang-def langs-def, fast*) — And thus c is in the language of the saturated automaton

qed

qed

5.4.3 Precision

In this section we show the precision of the algorithm, that is we show that the saturated language is below the backwards reachable set.

The following induction scheme makes an induction over the number of occurrences of a certain transition in words accepted by a FSM:

To prove a proposition for all words from state qs to state qf in FSM A that has a transition rule $(s, a, s') \in \delta\ A$, we have to show the following:

- Show, that the proposition is valid for words that do not use the transition rule $(s, a, s') \in \delta\ A$ at all
- Assuming that there is a prefix wp from qs to s and a suffix ws from s' to qf , and that wp does not use the new rule, and further assuming that for all prefixes wh from qs to s' , the proposition holds for $wh @ ws$, show that the proposition also holds for $wp @ a \# ws$.

We actually do use D here instead of δA , for use with $trclAD$.

lemma *ins-trans-induct*[*case-names base step*]:

fixes qs and qf
assumes $A: (qs, w, qf) \in trclAD A (insert (s, a, s') D)$
assumes *BASE-CASE*: $!! w . (qs, w, qf) \in trclAD A D \implies P w$
assumes *STEP-CASE*: $!! wp ws . [(qs, wp, s) \in trclAD A D; (s', ws, qf) \in trclAD A (insert (s, a, s') D)]; !! wh . (qs, wh, s') \in trclAD A D \implies P (wh@ws)] \implies P (wp@a\#ws)$
shows $P w$
proof –
– Essentially, the proof works by induction over the suffix ws
{
fix ws
have $!! qh wp . [(qs, wp, qh) \in trclAD A D; (qh, ws, qf) \in trclAD A (insert (s, a, s') D)] \implies P (wp@ws)$ **proof** (*induct ws*)
case (*Nil qh wp*) **with** *BASE-CASE* **show** $?case$ **by** (*subgoal-tac qh=qf, auto*)
next
case (*Cons e w qh wp*) **note** *IHP=this*
then obtain qhh **where** *SPLIT*: $(qh, e, qhh) \in (insert (s \hookrightarrow_a s') D) \wedge (qhh, w, qf) \in trclAD A (insert (s \hookrightarrow_a s') D) \wedge qh \in Q A \wedge e \in \Sigma A$ **by** (*fast dest: trclAD-uncons*)
show $?case$ **proof** (*cases (qh, e, qhh) = (s, a, s')*)
case *False*
with *SPLIT* **have** $(qh, [e], qhh) \in trclAD A D$ **by** (*auto intro: trclAD-one-elem dest: trclAD-elems*)
with *IHP* **have** $(qs, wp@[e], qhh) \in trclAD A D$ **by** (*fast intro: trclAD-concat*)
with *IHP SPLIT* **have** $P ((wp@[e])@w)$ **by** *fast*
thus $?thesis$ **by** *simp*
next
case *True* **note** *CASE=this*
with *SPLIT IHP* **have** $(qs, wp, s) \in trclAD A D \wedge s' \hookrightarrow_w qf \in trclAD A (insert (s \hookrightarrow_a s') D) !! wh . (qs, wh, s') \in trclAD A D \implies P (wh@w)$ **by** *simp-all*
with *STEP-CASE CASE* **show** $?thesis$ **by** *simp*
qed
qed
} **note** $C=this$
from $A C[of [] qs w]$ **show** $?thesis$ **by** (*auto dest: trclAD-elems*)
qed

The following lemma is a stronger elimination rule than *ps-R.cases*. It makes a more fine-grained distinction. In words: A step of the algorithm adds a transition $(sp A q p, \gamma, s')$, if there is a rule $([p, \gamma], a, p' \# c')$, and a transition sequence $(q, p' \# c', s') \in trclAD A D$. That is, if we have $(sp A q p', c', s') \in trclAD A D$.

lemma (*in MFSM*) *ps-R-elim-adv*:

$[(D, D') \in ps-R M A; !! \gamma s' a p' c' p q. [$
 $D' = insert (sp A q p, \gamma, s') D; (sp A q p, \gamma, s') \notin D; [p, \gamma] \hookrightarrow_a p' \# c' \in rules M;$
 $(q, p' \# c', s') \in trclAD A D;$
 $p \in csyms M; \gamma \in ssyms M; q \in cstates A; p' \in csyms M; a \in labels M; (q, p', sp A q$

```

 $p' \in D; (sp\ A\ q\ p', c', s') \in trclAD\ A\ D$ 
  ]  $\implies P$  ]
 $\implies P$ 
apply (erule ps-R.cases)
apply (auto simp add: uniqueSp)
proof –
  case (goal1 p  $\gamma$  a c' q s') note A=this
  then obtain p' cc' where RFMT:  $p \in csyms\ M \wedge c' = p' \# cc' \wedge p' \in csyms\ M \wedge$ 
 $\gamma \in ssyms\ M \wedge a \in labels\ M$  by (auto elim!: rule-cases)
  with A obtain gh where SPLIT:  $(q, p', gh) \in D \wedge (gh, cc', s') \in trclAD\ A\ D$  by
(fast dest: trclAD-uncons)
  with A RFMT have  $q \in cstates\ A \wedge gh = sp\ A\ q\ p'$  by (subgoal-tac  $(q, p', gh) \in ps\ upper$ 
 $M\ A$ ) (auto elim!: ps-upper-cases dest: syms-part sep-out-syms)
  with prems show ?case by fast
qed

```

Now follows a helper lemma to establish the precision result. In the original paper [1] it is called the *crucial point* of the precision proof.

It states that for transition relations that occur during the execution of the algorithm, for each word w that leads from the start state to a state $sp\ A\ q\ p$, there is a word $ws\ @\ [p]$ that leads to $sp\ A\ q\ p$ in the initial automaton and w can be rewritten to $ws\ @\ [p]$.

In the initial transition relation, a state of the form $sp\ A\ q\ p$ has only one incoming edge labelled p (*MFSM.sp-pred-ex* *MFSM.sp-pred-unique*). Intuitively, this lemma explains why it is correct to add further incoming edges to $sp\ A\ q\ p$: All words using such edges can be rewritten to a word using the original edge.

lemma (in *MFSM*) *sp-property*:

shows *is-inv* (ps-R *M* *A*) ($\delta\ A$) (λD).

$(\forall w . \forall p \in csyms\ M . \forall q \in cstates\ A . (s0\ A, w, sp\ A\ q\ p) \in trclAD\ A\ D \longrightarrow (\exists ws\ as . (s0\ A, ws, q) \in trclA\ A \wedge (w, as, ws@[p]) \in trcl\ (tr\ (rules\ M)))) \wedge$
 $(\forall P'. is-inv\ (ps-R\ M\ A)\ (\delta\ A)\ P' \longrightarrow P'\ D)$

— We show the thesis by proving that it is an invariant of the saturation procedure

apply (rule inv-useI)

apply (auto simp add: initial-delta-below)

proof –

— Base case, show the thesis for the initial automata

fix $w\ p\ q$

assume $A: p \in csyms\ M\ q \in cstates\ A\ s0\ A \xleftrightarrow{w} sp\ A\ q\ p \in trclA\ A$

show $\exists ws . s0\ A \xleftrightarrow{ws} q \in trclA\ A \wedge (\exists as . (w, as, ws@[p]) \in trcl\ (tr\ (rules\ M)))$

proof (cases w rule: rev-cases) — Make a case distinction whether w is empty

case *Nil* — w cannot be empty, because $s0$ is a control state, and sp is a stack state, and by definition of M-automata, these cannot be equal

with A **have** $s0\ A = sp\ A\ q\ p$ **by** (auto)

with A *s0-fmt* *uniqueSp* **have** *False* **by** (auto dest: mfsm-cons')

thus ?thesis ..

next
case (*snoc ws p'*) **note** *CASE=this*
with *A* **obtain** *qh* **where** $(s0\ A, ws, qh) \in trclA\ A \wedge (qh, [p'], sp\ A\ q\ p) \in trclA\ A$
 $\wedge (qh, p', sp\ A\ q\ p) \in \delta\ A$ **by** (*fast dest: trclAD-unconcat trclAD-single*) — Get the last state *qh* and symbol *p'* before reaching *sp*
moreover with *A* **have** $p=p' \wedge qh=q$ **by** (*blast dest: sp-pred-unique*) — This symbol is *p*, because the *p*-edge from *q* is the only edge to *sp A q p* in an M-automata
moreover with *CASE* **have** $(w, [], ws@[p]) \in trcl\ (tr\ (rules\ M))$ **by** (*fast intro: trcl.empty*)
ultimately show *?thesis* **by** (*blast*)
qed
next
— Step case
fix *D1 D2 w p q*
assume
IH: $\forall w. \forall p \in csyms\ M. \forall q \in cstates\ A. s0\ A \hookrightarrow_w\ sp\ A\ q\ p \in trclAD\ A\ D1 \longrightarrow$
 $(\exists ws. s0\ A \hookrightarrow_{ws}\ q \in trclAD\ A\ (\delta\ A) \wedge (\exists as. w \hookrightarrow_{as}\ ws\ @\ [p] \in trcl\ (tr\ (rules\ M))))$ — By induction hypothesis, our proposition is valid for *D1*
and *SUCC: $(D1, D2) \in ps-R\ M\ A$* — We have to show the proposition for some *D2*, that is a successor state of *D1* w.r.t. *ps-R M A*
and *P1: $p \in csyms\ M\ q \in cstates\ A$* **and** *P2: $s0\ A \hookrightarrow_w\ sp\ A\ q\ p \in trclAD\ A\ D2$* — Premise of our proposition: We reach some state *sp A q p*
and *USE-INV: $\bigwedge P'. is-inv\ (ps-R\ M\ A)\ (\delta\ A)\ P' \implies P'\ D1$* — We can use known invariants

from *SUCC* **have** *SS: $D1 \subseteq ps-upper\ M\ A$* **by** (*blast dest: ps-R-dom-below*)
from *USE-INV* **have** *A2: $\delta\ A \subseteq D1$* **by** (*blast intro: ps-R-above-inv*)

from *SUCC* **obtain** $\gamma\ s'\ pp\ aa\ cc'\ qq$ **where** *ADD: insert (sp A qq pp, γ, s') D1 = D2 \wedge (sp A qq pp, γ, s') \notin D1* **and**
RCONS: $([pp, \gamma], aa, cc') \in rules\ M \wedge (qq, cc', s') \in trclAD\ A\ D1 \wedge qq \in cstates\ A \wedge pp \in csyms\ M \wedge aa \in labels\ M$
by (*blast elim!: ps-R-elims-adv*) — Because of *SUCC*, we obtain *D2* by adding a (new) transition $(sp\ A\ qq\ pp, \gamma, s')$ to *D1*, such that there is a rule $([pp, \gamma], aa, cc') \in rules\ M$ and the former transition relation can do $(qq, cc', s') \in trclAD\ A\ D1$
from *P2* *ADD* **have** *P2': $s0\ A \hookrightarrow_w\ sp\ A\ q\ p \in trclAD\ A$* (*insert (sp A qq pp $\hookrightarrow_{\gamma}\ s')$ D1)* **by** *simp*

show $\exists ws. s0\ A \hookrightarrow_{ws}\ q \in trclA\ A \wedge (\exists as. w \hookrightarrow_{as}\ ws\ @\ [p] \in trcl\ (tr\ (rules\ M)))$ **using** *P2'*
— We show the proposition by induction on how often the new rule was used. For this, we regard a prefix until the first usage of the added rule, and a suffix that may use the added rule arbitrarily often
proof (*rule ins-trans-induct*)
case (*goal1 w*) — Base case, the added rule is not used at all. The proof is straightforward using the induction hypothesis of the outer (invariant) induction
thus *?case* **using** *IH P1* **by** *simp*

next

fix $wpre\ wsfx$ — Step case: We have a prefix that does not use the added rule, then a usage of the added rule and a suffix. We know that our proposition holds for all prefixes that do not use the added rule.

assume $IP1: (s0\ A, wpre, sp\ A\ qq\ pp) \in trclAD\ A\ D1$ **and** $IP2: (s', wsfx, sp\ A\ q\ p) \in trclAD\ A\ (insert\ (sp\ A\ qq\ pp, \gamma, s')\ D1)$

assume $IIIH: !!wh. (s0\ A, wh, s') \in trclAD\ A\ D1 \implies \exists ws. (s0\ A, ws, q) \in trclAD\ A\ (\delta\ A) \wedge (\exists as. (wh\ @\ wsfx, as, ws\ @\ [p]) \in trcl\ (tr\ (rules\ M)))$

from $IP1\ IH\ RCONS$ **obtain** $wps\ aps$ **where** $C1: (s0\ A, wps, qq) \in trclAD\ A\ (\delta\ A) \wedge wpre \hookrightarrow_{aps} wps\ @\ [pp] \in trcl\ (tr\ (rules\ M))$ **by** *fast* — This is an instance of a configuration reaching a sp-state, thus by induction hypothesis of the outer (invariant) induction, we find a successor configuration $wps\ @\ [pp]$ that reaches this state using pp as last edge in $\delta\ A$

with $A2$ **have** $(s0\ A, wps, qq) \in trclAD\ A\ D1$ **by** (*blast dest: trclAD-mono*) — And because $\delta\ A \subseteq D1$, we can do the transitions also in $D1$

with $RCONS$ **have** $(s0\ A, wps@cc', s') \in trclAD\ A\ D1$ **by** (*blast intro: trclAD-concat*) — From above ($RCONS$) we know $(qq, cc', s') \in trclAD\ A\ D1$, and we can concatenate these transition sequences

then obtain $ws\ as$ **where** $C2: (s0\ A, ws, q) \in trclAD\ A\ (\delta\ A) \wedge (wps@cc')\ @\ wsfx \hookrightarrow_{as} ws\ @\ [p] \in trcl\ (tr\ (rules\ M))$ **by** (*fast dest: IIIH*) — This concatenation is a prefix to a usage of the added transition, that does not use the added transition itself. (The whole configuration bringing us to $sp\ A\ q\ p$ is $wps\ @\ cc'\ @\ wsfx$). For those prefixes, we can apply the induction hypothesis of the inner induction and obtain a configuration $ws\ @\ [p]$ that is a successor configuration of $wps\ @\ cc'\ @\ wsfx$, and with which we can reach $sp\ A\ q\ p$ using p as last edge

have $\exists as. wpre\ @\ \gamma\ \# wsfx \hookrightarrow_{as} ws\ @\ [p] \in trcl\ (tr\ (rules\ M))$ — Now we obtained some configuration $ws\ @\ [p]$, that reaches $sp\ A\ q\ p$ using p as last edge in $\delta\ A$. Now we show that this is indeed a successor configuration of $wpre\ @\ \gamma\ \# wsfx$.

proof —

— This is done by putting together the transitions and using the extensibility of string rewrite systems, i.e. that we can still do a rewrite step if we add context

from $C1$ **have** $wpre@(\gamma\ \# wsfx) \hookrightarrow_{aps} (wps@[pp])@(\gamma\ \# wsfx) \in trcl\ (tr\ (rules\ M))$ **by** (*fast intro: srs-ext*)

hence $wpre@(\gamma\ \# wsfx) \hookrightarrow_{aps} wps@[pp, \gamma]@wsfx \in trcl\ (tr\ (rules\ M))$ **by** *simp*

moreover from $RCONS$ **have** $wps@[pp, \gamma]@wsfx \hookrightarrow_{[aa]} wps@cc'@wsfx \in trcl\ (tr\ (rules\ M))$ **by** (*fast intro: tr.rewrite trcl-one-elem*)

hence $wps@[pp, \gamma]@wsfx \hookrightarrow_{[aa]} (wps@cc')@wsfx \in trcl\ (tr\ (rules\ M))$ **by** *simp*

moreover note $C2$

ultimately have $wpre@(\gamma\ \# wsfx) \hookrightarrow_{aps@[aa]@as} ws@[p] \in trcl\ (tr\ (rules\ M))$

by (*fast intro: trcl-concat*)

thus *?thesis* **by** *fast*

qed

with $C2$ **show** $\exists ws. s0\ A \hookrightarrow_{ws} q \in trclAD\ A\ (\delta\ A) \wedge (\exists as. wpre\ @\ \gamma\ \# wsfx \hookrightarrow_{as} ws\ @\ [p] \in trcl\ (tr\ (rules\ M)))$ **by** *fast* — Finally, we have the proposition for the configuration $wpre\ @\ \gamma\ \# wsfx$, that contains the added rule (s, γ, s') one time more

qed
qed

Helper lemma to clarify some subgoal in the precision proof:

lemma *trclAD-delta-update-inv*: $trclAD (A(\delta:=X)) D = trclAD A D$ **by** (*unfold trclAD-def trclADp-def, simp*)

The precision is proved as an invariant of the saturation algorithm:

theorem (in *MFSM*) *precise-inv*:

shows $is-inv (ps-R M A) (\delta A) (\lambda D. (lang (A(\delta:=D))) \subseteq pre^* (rules M) A) \wedge (\forall P'. is-inv (ps-R M A) (\delta A) P' \longrightarrow P' D)$

apply (*rule inv-useI*)

— The base case is solved automatically, it follows from the reflexivity of pre^* .

apply (*simp, fast intro: pre-refl*)

apply (*unfold lang-def langs-def*) — Tune the step-case a bit before entering isar-proof

apply (*auto simp add: trclAD-delta-update-inv*)

proof —

fix $D1 D2 w f$

assume *IH*: $\{w. \exists f \in F A. s0 A \hookrightarrow_w f \in trclAD A D1\} \subseteq pre^* (rules M) A$ —

By induction hypothesis, we know $lang (A(\delta := D1)) \subseteq pre^* (rules M) A$

assume *SUCC*: $(D1, D2) \in ps-R M A$ — We regard a successor $D2$ of $D1$ w.r.t. $ps-R M A$

assume *P1*: $f \in F A$ **and** *P2*: $s0 A \hookrightarrow_w f \in trclAD A D2$ — And a word $w \in lang (A(\delta := D2))$

assume *USE-INV*: $\bigwedge P'. is-inv (ps-R M A) (\delta A) P' \implies P' D1$ — For the proof, we can use any known invariants

from *SUCC* **obtain** $\gamma s' p a c' q$ **where** *ADD*: $insert (sp A q p, \gamma, s') D1 = D2 \wedge (sp A q p, \gamma, s') \notin D1$ **and**

RCONS: $([p, \gamma], a, c') \in rules M \wedge (q, c', s') \in trclAD A D1 \wedge q \in cstates A \wedge p \in csyms M \wedge a \in labels M \wedge \gamma \in ssyms M$

by (*blast elim!: ps-R-elim-s-adv*) — Because of $(D1, D2) \in ps-R M A$, we obtain $D2$ by adding a (new) transition $(sp A q p, \gamma, s')$ to $D1$, such that there is a rule $([p, \gamma], a, c')$ and we have $(q, c', s') \in trclAD A D1$

from *P2* *ADD* **have** *P2'*: $s0 A \hookrightarrow_w f \in trclAD A (insert (sp A q p \hookrightarrow_\gamma s') D1)$

by *simp*

from *SUCC* **have** *SS*: $D1 \subseteq ps-upper M A$ **by** (*blast dest: ps-R-dom-below*) —

We know, that the intermediate value is below the upper saturation bound

from *USE-INV* **have** *A2*: $\delta A \subseteq D1$ **by** (*blast intro: ps-R-above-inv*) — ... and above the start value

from *SS* *USE-INV* *sp-property* **have** *SP-PROP*: $(\forall w. \forall p \in csyms M. \forall q \in cstates A. (s0 A, w, sp A q p) \in trclAD A D1 \longrightarrow (\exists ws as. (s0 A, ws, q) \in trclA A \wedge (w, as, ws@[p]) \in trcl (tr (rules M))))$

by *blast* — And we have just shown *sp-property*, that tells us that each configuration w that leads to a state $sp A q p$, can be rewritten to a configuration in the initial automaton, that uses p as its last transition

show $w \in pre^* (rules M) A$ **using** *P2'* — We have to show that the word w

from the new automaton is also in pre^* (rules M) A . We show this by induction on how often the new transition is used by w

proof (rule *ins-trans-induct*)

fix wa **assume** $(s0\ A, wa, f) \in trclAD\ A\ D1$ — Base case: w does not use the new transition at all

with $IH\ P1$ **show** $wa \in pre^*$ (rules M) A **by** (*fast*) — The proposition follows directly from the outer (invariant) induction and can be solved automatically

next

fix $wpre\ wsfx$ — Step case

assume $IP1$: $(s0\ A, wpre, sp\ A\ q\ p) \in trclAD\ A\ D1$ — We assume that we have a prefix $wpre$ leading to the start state s of the new transition and not using the new transition

assume $IP2$: $(s', wsfx, f) \in trclAD\ A\ (insert\ (sp\ A\ q\ p, \gamma, s')\ D1)$ — We also have a suffix from the end state s' to f

assume $IIIH$: $!!wh. (s0\ A, wh, s') \in trclAD\ A\ D1 \implies wh\ @\ wsfx \in pre^*$ (rules M) A — And we assume that our proposition is valid for prefixes wh that do not use the new transition

— We have to show that the proposition is valid for $wpre\ @\ \gamma\ \#\ wsfx$

from $IP1\ SP-PROP\ RCONS$ **obtain** $wpres\ apres$ **where** SPP : $(s0\ A, wpres, q) \in trclA\ A\ \wedge\ wpre \hookrightarrow_{apres}\ wpres@[p] \in trcl\ (tr\ (rules\ M))$ **by** (*blast*) — We can apply $SP-PROP$, to find a successor $wpres\ @\ [p]$ of $wpre$ in the initial automata

with $A2$ **have** $s0\ A \hookrightarrow_{wpres}\ q \in trclAD\ A\ D1$ **by** (*blast dest: trclAD-mono*) — $wpres$ can also be read by $D1$ because of $\delta\ A \subseteq D1$

with $RCONS$ **have** $s0\ A \hookrightarrow_{wpres@c'}\ s' \in trclAD\ A\ D1$ **by** (*fast intro: trclAD-concat*) — Altogether we get a prefix $wpres\ @\ c'$ that leads to s' , without using the added transition

with $IIIH$ **have** $(wpres@c')@wsfx \in pre\text{-}star\ (rules\ M)\ A$ **by** *fast* — We can apply the induction hypothesis

then obtain $as\ wo$ **where** $C1$: $wpres@c'@wsfx \hookrightarrow_{as}\ wo \in trcl\ (tr\ (rules\ M)) \wedge wo \in lang\ A$ **by** (*auto elim!: pre-star-altE*) — And find that there is a wo in the original automata, that is a successor of $wpres\ @\ c'\ @\ wsfx$

moreover have $\exists as. wpre@ \gamma \# wsfx \hookrightarrow_{as}\ wo \in trcl\ (tr\ (rules\ M))$ — Next we show that wo is a successor of $wpre\ @\ \gamma\ \#\ wsfx$

proof —

from SPP **have** $wpre@ \gamma \# wsfx \hookrightarrow_{apres}\ (wpres@[p])@ \gamma \# wsfx \in trcl\ (tr\ (rules\ M))$ **by** (*fast intro: srs-ext*)

hence $wpre@ \gamma \# wsfx \hookrightarrow_{apres}\ wpres@([p, \gamma])@wsfx \in trcl\ (tr\ (rules\ M))$ **by** *simp*

moreover from $RCONS$ **have** $wpres@([p, \gamma])@wsfx \hookrightarrow_{[a]}\ wpres@c'@wsfx \in trcl\ (tr\ (rules\ M))$ **by** (*fast intro: tr.rewrite trcl-one-elem*)

moreover note $C1$

ultimately show *?thesis* **by** (*fast intro: trcl-concat*)

qed

ultimately show $wpre\ @\ \gamma\ \#\ wsfx \in pre^*$ (rules M) A **by** (*fast intro: pre-star-altI*) — And altogether we have $wpre\ @\ \gamma\ \#\ wsfx \in pre^*$ (rules M) A

qed

qed

As precision is an invariant of the saturation algorithm, and is trivial for the case of an already saturated initial automata, the result of the saturation algorithm is precise

corollary (in *MFSM*) *precise*: $\llbracket (\delta A, D) \in \text{ndet-algo } (ps-R M A); x \in \text{lang } (A(\delta := D)) \rrbracket \implies x \in \text{pre-star } (rules M) A$
by (*auto elim!*: *ndet-algoE dest: inv intro: precise-inv pre-refl*)

And finally we get correctness of the algorithm, with no restrictions on valid states

theorem (in *MFSM*) *correct*: $\llbracket (\delta A, D) \in \text{ndet-algo } (ps-R M A) \rrbracket \implies \text{lang } (A(\delta := D)) = \text{pre-star } (rules M) A$ **by** (*auto intro: precise sound*)

So the main results of this theory are, that the algorithm is defined for every possible initial automata

$MFSM ?M ?A \implies \exists D. (\delta ?A, D) \in \text{ndet-algo } (ps-R ?M ?A)$

and returns the correct result

$\llbracket MFSM ?M ?A; (\delta ?A, ?D) \in \text{ndet-algo } (ps-R ?M ?A) \rrbracket \implies \text{lang } (?A(\delta := ?D)) = \text{pre}^* (rules ?M) ?A$

We could also prove determination, i.e. the terminating state is uniquely determined by the initial state (though there may be many ways to get there). This is not really needed here, because for correctness, we do not look at the structure of the final automaton, but just at its language. The language of the final automaton is determined, as implied by *MFSM.correct*.

end

6 Non-executable implementation of the DPN pre*-algorithm

theory *DPN-impl*
imports *DPN*
begin

This theory is to explore how to prove the correctness of straightforward implementations of the DPN pre* algorithm. It does not provide an executable specification, but uses set-datatype and the SOME-operator to describe a deterministic refinement of the nondeterministic pre*-algorithm. This refinement is then characterized as a recursive function, using recdef.

This proof uses the same techniques to get the recursive function and prove its correctness as are used for the straightforward executable implementation in *DPN_implEx*. Differences from the executable specification are:

- The state of the algorithm contains the transition relation that is saturated, thus making the refinement abstraction just a projection onto

this component. The executable specification, however, uses list representation of sets, thus making the refinement abstraction more complex.

- The termination proof is easier: In this approach, we only do recursion if our state contains a valid M-automata and a consistent transition relation. Using this property, we can infer termination easily from the termination of *ps-R*. The executable implementation does not check whether the state is valid, and thus may also do recursion for invalid states. Thus, the termination argument must also regard those invalid states, and hence must be more general.

6.1 Definitions

types $(c,l,s,m1,m2)$ *pss-state* = $((c,l,m1)$ *DPN-rec-scheme* * $(s,c,m2)$ *MFSM-rec-scheme*) * (s,c) *LTS*)

Function to select next transition to be added

constdefs

pss-isNext :: $(c,l,m1)$ *DPN-rec-scheme* \Rightarrow $(s,c,m2)$ *MFSM-rec-scheme* \Rightarrow (s,c) *LTS* \Rightarrow $(s*c*s)$ \Rightarrow *bool*
pss-isNext *M A D t* == $t \notin D \wedge (\exists q p \gamma q' a c'. t = (sp A q p, \gamma, q') \wedge [p, \gamma] \xrightarrow{a} c' \in rules M \wedge (q, c', q') \in trclAD A D)$
pss-next *M A D* == *if* $(\exists t. pss-isNext M A D t)$ *then Some (SOME t. pss-isNext M A D t)* *else None*

Next state selector function

constdefs

pss-next-state *S* == *case* *S* *of* $((M,A),D) \Rightarrow$ *if* *MFSM* *M A* $\wedge D \subseteq ps-upper M A$ *then* $(case pss-next M A D of None \Rightarrow None | Some t \Rightarrow Some ((M,A),insert t D))$ *else None*

Relation describing the deterministic algorithm

constdefs

pss-R == *graph* *pss-next-state*

lemma *pss-nextE1*: *pss-next* *M A D* = *Some t* $\implies t \notin D \wedge (\exists q p \gamma q' a c'. t = (sp A q p, \gamma, q') \wedge [p, \gamma] \xrightarrow{a} c' \in rules M \wedge (q, c', q') \in trclAD A D)$

proof –

assume *pss-next* *M A D* = *Some t*

hence *pss-isNext* *M A D t*

apply $(unfold pss-next-def)$

apply $(cases \exists t. pss-isNext M A D t)$

by $(auto intro: someI)$

thus *?thesis* **by** $(unfold pss-isNext-def)$

qed

lemma *pss-nextE2*: $pss\text{-next } M A D = None \implies \neg(\exists q p \gamma q' a c' t. t \notin D \wedge t = (sp A q p, \gamma, q') \wedge [p, \gamma] \xrightarrow{a} c' \in rules M \wedge (q, c', q') \in trclAD A D)$

proof –

assume $pss\text{-next } M A D = None$
hence $\neg(\exists t. pss\text{-isNext } M A D t)$
apply (*unfold pss-next-def*)
apply (*cases* $\exists t. pss\text{-isNext } M A D t$)
by *auto*
thus *?thesis* **by** (*unfold pss-isNext-def*) *blast*

qed

lemmas (**in** *MFSM*) $pss\text{-nextE} = pss\text{-nextE1 } pss\text{-nextE2}$

The relation of the deterministic algorithm is also the recursion relation of the recursive characterization of the algorithm

lemma *pss-R-alt[recdef-simp]*: $pss\text{-R} == \{((M, A), D), ((M, A), insert t D) \mid M A D t. MFSM M A \wedge D \subseteq ps\text{-upper } M A \wedge pss\text{-next } M A D = Some t\}$

by (*rule eq-reflection, unfold pss-R-def graph-def pss-next-state-def*) (*auto split: option.split-asm split-if-asm*)

6.2 Refining *ps-R*

We first show that the next-step relation refines *ps-R* $M A$. From this, we will get both termination and correctness

Abstraction relation to project on the second component of a tuple, with fixed first component

constdefs $\alpha snd f == \{ (s, (f, s)) \mid s. True \}$

lemma *$\alpha snd\text{-comp-simp}$* : $\alpha snd f O R = \{(s, (f, s')) \mid s s'. (s, s') \in R\}$ **by** (*unfold $\alpha snd\text{-def}$, blast*)

lemma *$\alpha sndI$ [simp]*: $(s, (f, s)) \in \alpha snd f$ **by** (*unfold $\alpha snd\text{-def}$, auto*)

lemma *$\alpha sndE$* : $(s, (f, s')) \in \alpha snd f' \implies f = f' \wedge s = s'$ **by** (*unfold $\alpha snd\text{-def}$, auto*)

Relation of *pss-next* and *ps-R* $M A$

lemma (**in** *MFSM*) *pss-cons1*: $\llbracket pss\text{-next } M A D = Some t; D \subseteq ps\text{-upper } M A \rrbracket \implies (D, insert t D) \in ps\text{-R } M A$ **by** (*auto dest: pss-nextE intro: ps-R.intros*)

lemma (**in** *MFSM*) *pss-cons2*: $pss\text{-next } M A D = None \implies D \notin Domain (ps\text{-R } M A)$ **by** (*blast dest: pss-nextE elim: ps-R.cases*)

lemma (**in** *MFSM*) *pss-cons1-rev*: $\llbracket D \subseteq ps\text{-upper } M A; D \notin Domain (ps\text{-R } M A) \rrbracket \implies pss\text{-next } M A D = None$ **by** (*cases pss-next M A D*) (*auto iff add: pss-cons1 pss-cons2*)

lemma (**in** *MFSM*) *pss-cons2-rev*: $\llbracket D \in Domain (ps\text{-R } M A) \rrbracket \implies \exists t. pss\text{-next } M A D = Some t \wedge (D, insert t D) \in ps\text{-R } M A$

by (*cases pss-next M A D*) (*auto iff add: pss-cons1 pss-cons2 ps-R-dom-below*)

The refinement result

theorem (in *MFSM*) *pss-refines*: $pss-R \leq_{\alpha snd} (M, A) (ps-R M A)$ **proof** (rule *refinesI*)
show $pss-R O \alpha snd (M, A) \subseteq \alpha snd (M, A) O ps-R M A$ **by** (rule *refines-compI*,
unfold $\alpha snd-def$ *pss-R-alt*) (*blast intro: pss-cons1*)
next
show $\alpha snd (M, A) \text{ “ Domain } (ps-R M A) \subseteq \text{Domain } pss-R$
apply (rule *refines-domI*)
apply (*unfold* $\alpha snd-def$ *pss-R-alt*)
apply (*unfold* *Domain-def*)
apply (*clarsimp*, *safe*)
apply (*simp add: prems*)
apply (*blast dest: ps-R-dom-below*)
apply (*insert pss-cons2-rev*, *fast*)
done
qed

6.3 Termination

We can infer termination directly from the well-foundedness of *ps-R* and *MFSM.pss-refines*

theorem *pss-R-wf[recdef-wf]*: $wf (pss-R^{-1})$ **proof** (rule *refines-wf[of pss-R snd*
 $\lambda r. \alpha snd (fst r) \lambda r. let (M, A) = fst r \text{ in } ps-R M A]$, *clarsimp*)
fix $M A D M' A' D'$
assume $A: (((M, A), D), ((M', A'), D')) \in pss-R$
then interpret *MFSM sep M M A*
apply (*unfold pss-R-alt MFSM-def*)
apply *blast*
apply *simp*
done
from *pss-refines ps-R-wf* **show** $pss-R \leq_{\alpha snd} (M, A) ps-R M A \wedge wf ((ps-R M A)^{-1})$ **by** *simp*
qed

6.4 Recursive characterization

Having proved termination, we can characterize our algorithm as a recursive function

consts

$pss-algo-rec :: (('c, 'l, 's, 'm1, 'm2) pss-state) \Rightarrow (('c, 'l, 's, 'm1, 'm2) pss-state)$

recdef *pss-algo-rec* $pss-R^{-1}$

$pss-algo-rec ((M, A), D) = (if (MFSM M A \wedge D \subseteq ps-upper M A) then (case (pss-next M A D) of None \Rightarrow ((M, A), D) | (Some t) \Rightarrow pss-algo-rec ((M, A), insert t D)) else ((M, A), D))$

lemma *pss-algo-rec-newsimps[simp]*:

$\llbracket MFSM M A; D \subseteq ps-upper M A; pss-next M A D = None \rrbracket \Longrightarrow pss-algo-rec ((M, A), D) = ((M, A), D)$

```

[[MFSM M A; D ⊆ ps-upper M A; pss-next M A D = Some t]] ⇒ pss-algo-rec
((M,A),D) = pss-algo-rec ((M,A),insert t D)
¬MFSM M A ⇒ pss-algo-rec ((M,A),D) = ((M,A),D)
¬(D ⊆ ps-upper M A) ⇒ pss-algo-rec ((M,A),D) = ((M,A),D)
by auto

```

```

declare pss-algo-rec.simps[simp del]

```

6.5 Correctness

The correctness of the recursive version of our algorithm can be inferred using the results from the locale *detRef-impl*

```

interpretation det-impl: detRef-impl pss-algo-rec pss-next-state pss-R
apply (rule detRef-impl.intro)
apply (simp-all add: detRef-wf-transfer[OF pss-R-wf] pss-R-def)
apply (unfold pss-next-state-def)
apply (auto split: split-if-asm)
apply (case-tac pss-next a b ba, auto)
apply (case-tac pss-next a b ba, auto)
apply (case-tac MFSM a b)
apply auto
apply (case-tac ba ⊆ ps-upper a b)
apply auto
done

```

```

theorem (in MFSM) pss-correct: lang (A | δ := snd (pss-algo-rec ((M,A),(δ A)))
|) = pre-star (rules M) A

```

```

proof -

```

```

have (((M,A),δ A), pss-algo-rec ((M,A),δ A)) ∈ ndet-algo pss-R by (rule det-impl.algo-correct)
moreover have (δ A, ((M,A),δ A)) ∈ αsnd (M,A) by simp
ultimately obtain D' where 1: (D', pss-algo-rec ((M,A),δ A)) ∈ αsnd (M,A)
and (δ A, D') ∈ ndet-algo (ps-R M A) using pss-refines by (blast dest: refines-ndet-algo)
with correct have lang (A | δ := D' |) = pre* (rules M) A by auto
moreover from 1 have snd (pss-algo-rec ((M,A),δ A)) = D' by (unfold αsnd-def,
auto)
ultimately show ?thesis by auto
qed

```

```

end

```

7 Tools for executable specifications

```

theory ImplHelper
imports Main
begin

```

7.1 Searching in Lists

Given a function f and a list l , return the result of the first element $e \in \text{set } l$ with $f e \neq \text{None}$. The functional code snippet `first-that f l` corresponds to the imperative code snippet: `for e in l do { if f e \neq None then return Some (f e) }; return None`

consts

`first-that :: ('s \Rightarrow 'a option) \Rightarrow 's list \Rightarrow 'a option`

primrec

`first-that f [] = None`

`first-that f (e#w) = (case f e of None \Rightarrow first-that f w | Some a \Rightarrow Some a)`

lemma `first-thatE1: first-that f l = Some a \implies $\exists e \in \text{set } l. f e = \text{Some } a$`

apply `(induct l)`

apply `simp`

apply `(case-tac f aa)`

apply `auto`

done

lemma `first-thatE2: first-that f l = None \implies $\forall e \in \text{set } l. f e = \text{None}$`

apply `(induct l)`

apply `simp`

apply `(case-tac f a)`

apply `auto`

done

lemmas `first-thatE = first-thatE1 first-thatE2`

lemma `first-thatI1: $e \in \text{set } l \wedge f e = \text{Some } a \implies \exists a'. \text{first-that } f l = \text{Some } a'$`

by `(cases first-that f l) (auto dest: first-thatE2)`

lemma `first-thatI2: $\forall e \in \text{set } l. f e = \text{None} \implies \text{first-that } f l = \text{None}$`

by `(cases first-that f l) (auto dest: first-thatE1)`

lemmas `first-thatI = first-thatI1 first-thatI2`

end

8 Executable algorithms for finite state machines

theory `FSM-ex`

imports `Main LTS FSM ImplHelper`

begin

The transition relation of a finite state machine is represented as a list of labeled edges

types `('s,'a) delta = ('s \times 'a \times 's) list`

8.1 Word lookup operation

Operation that finds some state q' that is reachable from state q with word w and has additional property P .

consts

$lookup :: ('s \Rightarrow bool) \Rightarrow ('s, 'a) \text{ delta} \Rightarrow 's \Rightarrow 'a \text{ list} \Rightarrow 's \text{ option}$

primrec

$lookup P d q [] = (if P q then Some q else None)$

$lookup P d q (e\#w) = \text{first-that } (\lambda t. \text{let } (qs, es, q') = t \text{ in if } q = qs \wedge e = es \text{ then } lookup P d q' w \text{ else None}) d$

lemma lookupE1: $!!q. lookup P d q w = Some q' \Longrightarrow P q' \wedge (q, w, q') \in \text{trcl } (set d)$ **proof** (induct w)

case Nil thus ?case by (cases P q) simp-all

next

case (Cons e w) note IHP=this

hence first-that $(\lambda t. \text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } lookup P d qh w \text{ else None}) d = Some q'$ **by** simp

then obtain t where $t \in set d \wedge ((\text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } lookup P d qh w \text{ else None}) = Some q')$ **by** (blast dest: first-thatE1)

then obtain qh where $(q, e, qh) \in set d \wedge lookup P d qh w = Some q'$

apply (cases t) **apply** simp **apply** (case-tac $q = a \wedge e = b$) **by** (auto)

moreover with IHP have $P q' \wedge (qh, w, q') \in \text{trcl } (set d)$ **by** auto

ultimately show ?case by auto

qed

lemma lookupE2: $!!q. lookup P d q w = None \Longrightarrow \neg(\exists q'. (P q') \wedge (q, w, q') \in \text{trcl } (set d))$ **proof** (induct w)

case Nil thus ?case by (cases P q) (auto dest: trcl-empty-cons)

next

case (Cons e w) note IHP=this

hence first-that $(\lambda t. \text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } lookup P d qh w \text{ else None}) d = None$ **by** simp

hence $\forall t \in set d. (\text{let } (qs, es, qh) = t \text{ in if } q = qs \wedge e = es \text{ then } lookup P d qh w \text{ else None}) = None$ **by** (blast dest: first-thatE2)

hence 1: $!! qs es qh. (qs, es, qh) \in set d \Longrightarrow q \neq qs \vee e \neq es \vee lookup P d qh w = None$ **by** auto

show ?case proof (rule ccontr, clarsimp)

fix q'

assume $C: P q' \wedge (q, e\#w, q') \in \text{trcl } (set d)$

then obtain qh where 2: $(q, e, qh) \in set d \wedge (qh, w, q') \in \text{trcl } (set d)$ **by** (blast dest: trcl-uncons)

with 1 have $lookup P d qh w = None$ **by** auto

with C 2 IHP show False **by** auto

qed

qed

lemma lookupI1: $\llbracket P q'; (q, w, q') \in \text{trcl } (set d) \rrbracket \Longrightarrow \exists q'. lookup P d q w = Some q'$

by (cases lookup $P d q w$) (auto dest: lookupE2)

lemma lookupI2: $\neg(\exists q'. P q' \wedge (q, w, q') \in \text{trcl } (\text{set } d)) \implies \text{lookup } P d q w = \text{None}$
by (cases lookup $P d q w$) (auto dest: lookupE1)

lemmas lookupE = lookupE1 lookupE2

lemmas lookupI = lookupI1 lookupI2

lemma lookup-trclAD-E1:

assumes map: set $d = D$ **and** start: $q \in Q A$ **and** cons: $D \subseteq Q A \times \Sigma A \times Q A$

assumes A: lookup $P d q w = \text{Some } q'$

shows $P q' \wedge (q, w, q') \in \text{trclAD } A D$

proof –

from A map **have** 1: $P q' \wedge (q, w, q') \in \text{trcl } D$ **by** (blast dest: lookupE1)

hence $(q, w, q') \in \text{trcl } (D \cap (Q A \times \Sigma A \times Q A)) \cap (Q A \times \text{UNIV} \times \text{UNIV})$

using cons start **by** (subgoal-tac $D = D \cap (Q A \times \Sigma A \times Q A)$, auto)

with 1 trclAD-by-trcl' **show** ?thesis **by** auto

qed

lemma lookup-trclAD-E2:

assumes map: set $d = D$

assumes A: lookup $P d q w = \text{None}$

shows $\neg(\exists q'. P q' \wedge (q, w, q') \in \text{trclAD } A D)$

proof –

from map A **have** $\neg(\exists q'. P q' \wedge (q, w, q') \in \text{trcl } D)$ **by** (blast dest: lookupE2)

with trclAD-subset-trcl **show** ?thesis **by** auto

qed

lemma lookup-trclAD-I1: $\llbracket \text{set } d = D; (q, w, q') \in \text{trclAD } A D; P q' \rrbracket \implies \exists q'. \text{lookup } P d q w = \text{Some } q'$

apply (cases lookup $P d q w$)

apply (subgoal-tac $\neg(\exists q'. P q' \wedge (q, w, q') \in \text{trclAD } A D)$)

apply simp

apply (rule lookup-trclAD-E2)

apply auto

done

lemma lookup-trclAD-I2: $\llbracket \text{set } d = D; q \in Q A; D \subseteq Q A \times \Sigma A \times Q A; \neg(\exists q'. P q' \wedge (q, w, q') \in \text{trclAD } A D) \rrbracket \implies \text{lookup } P d q w = \text{None}$

apply (cases lookup $P d q w$, auto)

apply (subgoal-tac $P a \wedge (q, w, a) \in \text{trclAD } A (\text{set } d)$)

apply blast

apply (rule lookup-trclAD-E1)

apply auto

done

lemmas lookup-trclAD-E = lookup-trclAD-E1 lookup-trclAD-E2

lemmas *lookup-trclAD-I = lookup-trclAD-I1 lookup-trclAD-I2*

8.2 Reachable states and alphabet inferred from transition relation

constdefs

states d == fst ‘ (set d) ∪ (snd ∘ snd) ‘ (set d)
alpha d == (fst ∘ snd) ‘ (set d)

lemma *statesAlphaI: (q,a,q') ∈ set d ⇒ q ∈ states d ∧ q' ∈ states d ∧ a ∈ alpha d*
by (*unfold states-def alpha-def, force*)

lemma *statesE: q ∈ states d ⇒ ∃ a q'. ((q,a,q') ∈ set d ∨ (q',a,q) ∈ set d)* **by** (*unfold states-def alpha-def, force*)

lemma *alphaE: a ∈ alpha d ⇒ ∃ q q'. (q,a,q') ∈ set d* **by** (*unfold states-def alpha-def, force*)

lemma *states-finite: finite (states d)* **by** (*unfold states-def, auto*)

lemma *alpha-finite: finite (alpha d)* **by** (*unfold alpha-def, auto*)

lemma *statesAlpha-subset: set d ⊆ states d × alpha d × states d* **by** (*auto dest: statesAlphaI*)

lemma *states-mono: set d ⊆ set d' ⇒ states d ⊆ states d'* **by** (*unfold states-def, auto*)

lemma *alpha-mono: set d ⊆ set d' ⇒ alpha d ⊆ alpha d'* **by** (*unfold alpha-def, auto*)

lemma *statesAlpha-insert: set d' = insert (q,a,q') (set d) ⇒ states d' = states d ∪ {q,q'} ∧ alpha d' = insert a (alpha d)*
by (*unfold states-def alpha-def*) (*simp, blast*)

lemma *statesAlpha-inv: [q ∈ states d; a ∈ alpha d; q' ∈ states d; set d' = insert (q,a,q') (set d)] ⇒ states d = states d' ∧ alpha d = alpha d'*
by (*unfold states-def alpha-def*) (*simp, blast*)

code-module *FSM-ex* **file** *FSM-ex.sml*

contains

lookup

end

9 Implementation of DPN pre*-algorithm

theory *DPN-implEx*

imports *DPN FSM-ex*

begin

In this section, we provide a straightforward executable specification of the DPN-algorithm. It has a polynomial complexity, but is far from having optimal complexity.

9.1 Representation of DPN and M-automata

types

'c rule-ex = 'c × 'c × 'c × 'c list
'c DPN-ex = 'c rule-ex list

constdefs

rule-repr == { ((p,γ,p',c'),(p#[γ],a,p'#c')) | p γ p' c' a . True }
rules-repr == { (l,l') . rule-repr “ set l = l' }

lemma *rules-repr-cons: [(R,S) ∈ rules-repr] ==> ((p,γ,p',c') ∈ set R) = (∃ a. (p#[γ] ↦_a p'#c') ∈ S)*

by *(unfold rules-repr-def rule-repr-def) blast*

We define the mapping to sp-states explicitly, well-knowing that it makes the algorithm even more inefficient

constdefs

find-sp d s p == first-that (λt. let (sh,ph,qh)=t in if s=sh ∧ p=ph then Some qh else None) d

This locale describes an M-automata together with its representation used in the implementation

locale *MFSM-ex = MFSM +*
fixes *R and D*

assumes *rules-repr: (R,rules M) ∈ rules-repr*

assumes *D-above: δ A ⊆ set D and D-below: set D ⊆ ps-upper M A*

This lemma exports the additional conditions of locale *MFSM-ex* to locale *MFSM*

lemma *(in MFSM) MFSM-ex-alt: MFSM-ex M A R D == (R,rules M) ∈ rules-repr ∧ δ A ⊆ set D ∧ set D ⊆ ps-upper M A*

using *prems by (unfold MFSM-def MFSM-ex-def MFSM-ex-axioms-def) (auto)*

lemmas *(in MFSM-ex) D-between = D-above D-below*

The representation of the sp-states behaves as expected

lemma *(in MFSM-ex) find-sp-cons:*

assumes *A: s ∈ cstates A p ∈ csyms M*

shows *find-sp D s p = Some (sp A s p)*

proof –

let *?f = (λt. let (sh,ph,qh)=t in if s=sh ∧ p=ph then Some qh else None)*

from *A have (s,p,sp A s p) ∈ set D using cstate-succ-ex' D-between by simp*

moreover have *?f (s,p,sp A s p) = Some (sp A s p) by auto*

ultimately obtain sp' **where** $G: \text{find-sp } D \text{ s } p = \text{Some } sp'$
using $\text{first-that}I1[\text{of } (s,p,sp \text{ A } s \text{ p}) \text{ D } ?f \text{ sp } A \text{ s } p]$ **by** $(\text{unfold find-sp-def}, \text{blast})$
with $\text{first-that}E1[\text{of } ?f \text{ D } sp']$ **obtain** t **where** $t \in \text{set } D \wedge ?f \text{ t} = \text{Some } sp'$ **by**
 $(\text{unfold find-sp-def}, \text{blast})$
hence $(s,p,sp') \in \text{set } D$ **by** $(\text{cases } t, \text{auto split: split-if-asm})$
with $A \text{ D-between}$ **have** $sp' = sp \text{ A } s \text{ p}$ **using** $\text{cstate-succ-unique}'$ **by** simp
with G **show** $?thesis$ **by** simp
qed

9.2 Next-element selection

The implementation goes straightforward by implementing a function to return the next transition to be added to the transition relation of the automata being saturated

constdefs

```

sel-next:: 'c DPN-ex  $\Rightarrow$  ('s,'c) delta  $\Rightarrow$  ('s  $\times$  'c  $\times$  's) option
sel-next R D ==
  first-that ( $\lambda r$ . let ( $p,\gamma,p',c'$ ) = r in
    first-that ( $\lambda t$ . let ( $q,pp',sp'$ ) = t in
      if  $pp' = p'$  then
        case find-sp D q p of
          Some  $spt \Rightarrow$  (case lookup ( $\lambda q'$ . ( $spt,\gamma,q'$ )  $\notin$  set D) D  $sp' \text{ c}'$  of
            Some  $q' \Rightarrow$  Some ( $spt,\gamma,q'$ ) |
            None  $\Rightarrow$  None
          ) | -  $\Rightarrow$  None
        else None
      ) D
    ) R

```

The state of our algorithm consists of a representation of the DPN-rules and a representation of the transition relations of the automata being saturated

types ('c,'s) *seln-state* = 'c DPN-ex \times ('s,'c) delta

As long as the next-element function returns elements, these are added to the transition relation and the algorithm is applied recursively. *sel-next-state* describes the next-state selector function, and *seln-R* describes the corresponding recursion relation.

constdefs

```

sel-next-state S == let (R,D)=S in case sel-next R D of None  $\Rightarrow$  None | Some t  $\Rightarrow$  Some (R,t#D)

```

constdefs

```

seln-R == graph sel-next-state

```

lemma *seln-R-alt[recdef-simp]*: *seln-R* == {((R,D),(R,t#D)) | R D t. *sel-next* R D = Some t}

by $(\text{rule eq-reflection}, \text{unfold seln-R-def graph-def sel-next-state-def})$ $(\text{auto split: option.split-asm})$

9.3 Termination

9.3.1 Saturation upper bound

Before we can define the algorithm as recursive function, we have to prove termination, that is well-foundedness of the corresponding recursion relation $\text{seln-}R$

We start by defining a trivial finite upper bound for the saturation, simply as the set of all possible transitions in the automata. Intuitively, this bound is valid because the saturation algorithm only adds transitions, but never states to the automata

constdefs

$\text{seln-triv-upper } R D == \text{states } D \times ((\text{fst} \circ \text{snd}) ' (\text{set } R) \cup \text{alpha } D) \times \text{states } D$

lemma $\text{seln-triv-upper-finite}$: $\text{finite } (\text{seln-triv-upper } R D)$ **by** $(\text{unfold } \text{seln-triv-upper-def})$
 $(\text{auto simp add: states-finite alpha-finite})$

lemma $D\text{-below-triv-upper}$: $\text{set } D \subseteq \text{seln-triv-upper } R D$ **using** $\text{statesAlpha-subset}$

by $(\text{unfold } \text{seln-triv-upper-def})$ auto

lemma $\text{seln-triv-upper-subset-preserve}$: $\text{set } D \subseteq \text{seln-triv-upper } A D' \implies \text{seln-triv-upper } A D \subseteq \text{seln-triv-upper } A D'$

by $(\text{unfold } \text{seln-triv-upper-def})$ $(\text{blast intro: statesAlphaI dest: statesE alphaE})$

lemma $\text{seln-triv-upper-mono}$: $\text{set } D \subseteq \text{set } D' \implies \text{seln-triv-upper } R D \subseteq \text{seln-triv-upper } R D'$

by $(\text{unfold } \text{seln-triv-upper-def})$ $(\text{auto dest: states-mono alpha-mono})$

lemma $\text{seln-triv-upper-mono-list}$: $\text{seln-triv-upper } R D \subseteq \text{seln-triv-upper } R (t\#D)$
by $(\text{auto intro!: seln-triv-upper-mono})$

lemma $\text{seln-triv-upper-mono-list}'$: $x \in \text{seln-triv-upper } R D \implies x \in \text{seln-triv-upper } R (t\#D)$ **using** $\text{seln-triv-upper-mono-list}$ **by** (fast)

The trivial upper bound is not changed by inserting a transition to the automata that was already below the upper bound

lemma $\text{seln-triv-upper-inv}$: $\llbracket t \in \text{seln-triv-upper } R D; \text{set } D' = \text{insert } t (\text{set } D) \rrbracket \implies \text{seln-triv-upper } R D = \text{seln-triv-upper } R D'$

by $(\text{unfold } \text{seln-triv-upper-def})$ $(\text{auto dest: statesAlpha-insert})$

States returned by find-sp are valid states of the underlying automaton

lemma find-sp-in-states : $\text{find-sp } D s p = \text{Some } qh \implies qh \in \text{states } D$ **by** $(\text{unfold } \text{find-sp-def})$ $(\text{auto dest: first-thatE1 split: split-if-asm simp add: statesAlphaI})$

The next-element selection function returns a new transition, that is below the trivial upper bound

lemma sel-next-below :

```

assumes A: sel-next R D = Some t

shows  $t \notin \text{set } D \wedge t \in \text{seln-triv-upper } R D$ 
using A
apply (unfold sel-next-def seln-triv-upper-def)
apply (clarsimp dest!: first-thatE1 lookupE1 split: split-if-asm option.split-asm)
proof (intro conjI)
  fix q a qh b q'
  assume A:  $(q, a, qh) \in \text{set } D$  and B:  $(qh, b, q') \in \text{trcl } (\text{set } D)$ 
  from B statesAlpha-subset[of D] show  $q' \in \text{states } D$ 
  apply –
  apply (erule (1) trcl-structE)
  apply (insert A)
  apply (simp add: statesAlphaI)
  by assumption
qed (force simp add: find-sp-in-states)+

```

Hence, it does not change the upper bound

```

corollary sel-next-upper-preserve:  $\llbracket \text{sel-next } R D = \text{Some } t \rrbracket \implies \text{seln-triv-upper } R D = \text{seln-triv-upper } R (t \# D)$  proof –
  have  $\text{set } (t \# D) = \text{insert } t (\text{set } D)$  by auto
  moreover assume sel-next R D = Some t
  with sel-next-below have  $t \in \text{seln-triv-upper } R D$  by blast
  ultimately show ?thesis by (blast dest: seln-triv-upper-inv)
qed

```

9.3.2 Well-foundedness of recursion relation

```

lemma seln-R-wf[recdef-wf]: wf (seln-R-1) proof –
  let ?rel =  $\{(R, D), (R, D') \mid R D D'. \text{set } D \subset \text{set } D' \wedge \text{seln-triv-upper } R D = \text{seln-triv-upper } R D'\}$ 
  have  $\text{seln-R}^{-1} \subseteq ?rel^{-1}$ 
  apply (unfold seln-R-alt)
  apply (clarsimp, safe)
  apply (blast dest: sel-next-below)
  apply (simp add: seln-triv-upper-mono-list')
  apply (simp add: sel-next-upper-preserve)
  done
also
  let ?alpha =  $\lambda x. \text{let } (R, D) = x \text{ in } \text{seln-triv-upper } R D - \text{set } D$ 
  let ?rel2 = finite-psubset-1
  have  $?rel^{-1} \subseteq \text{inv-image } (?rel2^{-1}) ?alpha$  using D-below-triv-upper by (unfold finite-psubset-def, fastsimp simp add: inv-image-def seln-triv-upper-finite)
  finally have  $\text{seln-R}^{-1} \subseteq \text{inv-image } (?rel2^{-1}) ?alpha$  .
  moreover
  have wf (?rel2-1) using wf-finite-psubset by simp
  hence wf ( $\text{inv-image } (?rel2^{-1}) ?alpha$ ) by (rule wf-inv-image)
  ultimately show ?thesis by (blast intro: wf-subset)
qed

```

9.3.3 Definition of recursive function

consts

$pss\text{-algo-rec} :: ('c,'s) \text{ seln-state} \Rightarrow ('c,'s) \text{ seln-state}$

recdef $pss\text{-algo-rec seln-R}^{-1}$

$pss\text{-algo-rec } (R,D) = (\text{case sel-next } R \ D \ \text{of Some } t \Rightarrow pss\text{-algo-rec } (R,t\#D) \mid \text{None} \Rightarrow (R,D))$

lemma $pss\text{-algo-rec-newsimps[simp]}$:

$\llbracket \text{sel-next } R \ D = \text{None} \rrbracket \Longrightarrow pss\text{-algo-rec } (R,D) = (R,D)$

$\llbracket \text{sel-next } R \ D = \text{Some } t \rrbracket \Longrightarrow pss\text{-algo-rec } (R,D) = pss\text{-algo-rec } (R,t\#D)$

by *auto*

declare $pss\text{-algo-rec.simps[simp del]}$

9.4 Correctness

9.4.1 seln_R refines ps_R

We show that seln-R refines ps-R , that is that every step made by our implementation corresponds to a step in the nondeterministic algorithm, that we already have proved correct in theory DPN.

lemma (in $MFSM\text{-ex}$) sel-nextE1 :

assumes $A: \text{sel-next } R \ D = \text{Some } (s,\gamma,q')$

shows $(s,\gamma,q') \notin \text{set } D \wedge (\exists q \ p \ a \ c'. s = sp \ A \ q \ p \wedge [p,\gamma] \xrightarrow{a} c' \in \text{rules } M \wedge (q,c',q') \in \text{trclAD } A \ (\text{set } D))$

proof –

let $?f = \lambda p \ \gamma \ p' \ c' \ t. \text{let } (q,pp',sp') = t \text{ in}$

if $pp' = p'$ *then*

case $\text{find-sp } D \ q \ p \ \text{of}$

$\text{Some } s \Rightarrow (\text{case lookup } (\lambda q'. (s,\gamma,q') \notin \text{set } D) \ D \ sp' \ c' \ \text{of}$

$\text{Some } q' \Rightarrow \text{Some } (s,\gamma,q') \mid$

$\text{None} \Rightarrow \text{None}$

$) \mid - \Rightarrow \text{None}$

else None

let $?f1 = \lambda r. \text{let } (p,\gamma,p',c') = r \ \text{in } \text{first-that } (?f \ p \ \gamma \ p' \ c') \ D$

from $A[\text{unfolded sel-next-def}]$ **obtain** r **where** $1: r \in \text{set } R \wedge ?f1 \ r = \text{Some } (s,\gamma,q')$ **by** (*blast dest: first-thatE1*)

then obtain $p \ \gamma h \ p' \ c'$ **where** $2: r = (p,\gamma h,p',c') \wedge \text{first-that } (?f \ p \ \gamma h \ p' \ c') \ D = \text{Some } (s,\gamma,q')$ **by** (*cases r, simp, blast*)

then obtain t **where** $3: t \in \text{set } D \wedge ?f \ p \ \gamma h \ p' \ c' \ t = \text{Some } (s,\gamma,q')$ **by** (*blast dest: first-thatE1*)

then obtain $q \ sp'$ **where** $4: t = (q,p',sp') \wedge (\text{case find-sp } D \ q \ p \ \text{of}$

$\text{Some } s \Rightarrow (\text{case lookup } (\lambda q'. (s,\gamma h,q') \notin \text{set } D) \ D \ sp' \ c' \ \text{of}$

$\text{Some } q' \Rightarrow \text{Some } (s,\gamma h,q') \mid$

$\text{None} \Rightarrow \text{None}$

) | - \Rightarrow None) = Some (s, γ , q')
 by (cases t, auto split: split-if-asm)
 hence 5: find-sp D q p = Some s \wedge lookup ($\lambda q'. (s, \gamma h, q') \notin \text{set } D$) D sp' c' =
 Some q' \wedge $\gamma = \gamma h$
 by (simp split: option.split-asm)
 with 1 2 rules-repr obtain a where 6: (p#[γ], a, p'#c') \in rules M by (blast dest:
 rules-repr-cons)
 hence 7: p \in csyms M \wedge p' \in csyms M \wedge $\gamma \in$ ssyms M by (blast dest: rule-fmt-fs)
 with 3 4 D-below have 8: q \in cstates A \wedge sp' = sp A q p' by (blast dest: csym-from-cstate'
 cstate-succ-unique')
 with 5 7 have 9: s = sp A q p using D-above D-below by (auto simp add:
 find-sp-cons)
 have 10: (s, γ , q') \notin set D \wedge (sp', c', q') \in trclAD A (set D) using 5 8 uniqueSp 7
 states-part D-below ps-upper-below-trivial
 apply – apply (rule lookup-trclAD-E1)
 by auto
 moreover have (q, p'#c', q') \in trclAD A (set D) proof –
 from 7 8 sp-pred-ex D-above have (q, p', sp') \in set D by auto
 with 10 trclAD.cons show ?thesis using 7 8 alpha-cons states-part by auto
 qed
 ultimately show ?thesis using 9 6 by blast
 qed

lemma (in MFSM-ex) sel-nextE2:

assumes A: sel-next R D = None
 shows $\neg(\exists q p \gamma q' a c' t. t \notin \text{set } D \wedge t = (sp A q p, \gamma, q') \wedge [p, \gamma] \hookrightarrow_a c' \in \text{rules } M$
 $\wedge (q, c', q') \in \text{trclAD } A \text{ (set } D))$
 proof (clarify) — Assume we have such a rule and transition, and infer sel-next R
 D \neq None
 fix q p γ q' a pc'
 assume C: (sp A q p, γ , q') \notin set D ([p, γ], a, pc') \in rules M (q, pc', q') \in
 trclAD A (set D)

from C obtain p' c' where SYMS: p \in csyms M \wedge p' \in csyms M \wedge $\gamma \in$ ssyms M
 \wedge pc' = p'#c' by (blast dest: rule-fmt)

have QCS: q \in cstates A (q, p', sp A q p') \in set D (sp A q p', c', q') \in trclAD A (set
 D) proof –

from C SYMS obtain sp' where (q, p', sp') \in set D \wedge (sp', c', q') \in trclAD A (set
 D) by (blast dest: trclAD-uncons)

moreover with D-below SYMS show q \in cstates A by (auto intro: csym-from-cstate')

ultimately show (q, p', sp A q p') \in set D (sp A q p', c', q') \in trclAD A (set D)

using D-below cstate-succ-unique' by auto

qed

from C QCS lookup-trclAD-I1 [of D set D sp A q p' c' q' A ($\lambda q''. (sp A q p, \gamma, q'')$
 \notin set D)] obtain q'' where N1: lookup ($\lambda q''. (sp A q p, \gamma, q'')$ \notin set D) D (sp A
 q p') c' = Some q'' by blast

let $?f = \lambda p \ \gamma \ p' \ c' \ q \ pp' \ sp'.$
 if $pp'=p'$ **then**
 case $find\text{-}sp \ D \ q \ p$ **of**
 Some $s \Rightarrow$ (**case** $lookup \ (\lambda q'. \ (s,\gamma,q') \notin set \ D)$ $D \ sp' \ c'$ **of**
 Some $q' \Rightarrow$ **Some** (s,γ,q') |
 None \Rightarrow **None**
) | $- \Rightarrow$ **None**
 else **None**

from $SYMS \ QCS$ **have** $FIND\text{-}SP:$ $find\text{-}sp \ D \ q \ p = Some \ (sp \ A \ q \ p)$ **using**
 $D\text{-below} \ D\text{-above}$ **by** ($simp \ add:$ $find\text{-}sp\text{-}cons$)
let $?f1 = (\lambda p \ \gamma \ p' \ c'. \ (\lambda t. \ let \ (q,pp',sp')=t \ in \ ?f \ p \ \gamma \ p' \ c' \ q \ pp' \ sp'))$
from $N1 \ FIND\text{-}SP$ **have** $N2:$ $?f1 \ p \ \gamma \ p' \ c' \ (q,p',sp \ A \ q \ p') = Some \ (sp \ A \ q \ p,$
 $\gamma, \ q')$ **by** $auto$
with $QCS \ first\text{-}thatI1$ [$of \ (q,p',sp \ A \ q \ p') \ D \ ?f1 \ p \ \gamma \ p' \ c'$] **obtain** t' **where** $N3:$
 $first\text{-}that \ (?f1 \ p \ \gamma \ p' \ c') \ D = Some \ t'$ **by** ($blast$)
let $?f2 = (\lambda r. \ let \ (p,\gamma,p',c') = r \ in \ first\text{-}that \ (?f1 \ p \ \gamma \ p' \ c') \ D)$
from $N3$ **have** $?f2 \ (p,\gamma,p',c') = Some \ t'$ **by** $auto$
moreover **from** $SYMS \ C \ rules\text{-}repr$ **have** $(p,\gamma,p',c') \in set \ R$ **by** ($blast \ dest:$
 $rules\text{-}repr\text{-}cons$)
ultimately **obtain** t'' **where** $first\text{-}that \ ?f2 \ R = Some \ t''$ **using** $first\text{-}thatI1$ [of
 $(p, \ \gamma, \ p', \ c') \ R \ ?f2$] **by** ($blast$)
hence $sel\text{-}next \ R \ D = Some \ t''$ **by** ($unfold \ sel\text{-}next\text{-}def$)
with A **show** $False$ **by** $simp$
qed

lemmas (**in** $MFSM\text{-}ex$) $sel\text{-}nextE = sel\text{-}nextE1 \ sel\text{-}nextE2$

lemma (**in** $MFSM\text{-}ex$) $seln\text{-}cons1:$ $\llbracket sel\text{-}next \ R \ D = Some \ t \rrbracket \Longrightarrow (set \ D, insert \ t \ (set \ D)) \in ps\text{-}R \ M \ A$ **using** $D\text{-below}$ **by** ($cases \ t, \ auto \ dest:$ $sel\text{-}nextE \ intro:$ $ps\text{-}R.intros$)
lemma (**in** $MFSM\text{-}ex$) $seln\text{-}cons2:$ $sel\text{-}next \ R \ D = None \Longrightarrow set \ D \notin Domain \ (ps\text{-}R \ M \ A)$ **by** ($blast \ dest:$ $sel\text{-}nextE \ elim:$ $ps\text{-}R.cases$)

lemma (**in** $MFSM\text{-}ex$) $seln\text{-}cons1\text{-}rev:$ $\llbracket set \ D \notin Domain \ (ps\text{-}R \ M \ A) \rrbracket \Longrightarrow sel\text{-}next \ R \ D = None$ **by** ($cases \ sel\text{-}next \ R \ D$) ($auto \ iff \ add:$ $seln\text{-}cons1 \ seln\text{-}cons2$)
lemma (**in** $MFSM\text{-}ex$) $seln\text{-}cons2\text{-}rev:$ $\llbracket set \ D \in Domain \ (ps\text{-}R \ M \ A) \rrbracket \Longrightarrow \exists t. \ sel\text{-}next \ R \ D = Some \ t \wedge (set \ D, insert \ t \ (set \ D)) \in ps\text{-}R \ M \ A$
by ($cases \ sel\text{-}next \ R \ D$) ($auto \ iff \ add:$ $seln\text{-}cons1 \ seln\text{-}cons2 \ ps\text{-}R\text{-}dom\text{-}below$)

DPN-specific abstraction relation, to associate states of deterministic algorithm with states of $ps\text{-}R$

constdefs $\alpha seln \ M \ A == \{ (set \ D, (R,D)) \mid D \ R. \ MFSM\text{-}ex \ M \ A \ R \ D \}$

lemma $\alpha selnI:$ $\llbracket S=set \ D; \ MFSM\text{-}ex \ M \ A \ R \ D \rrbracket \Longrightarrow (S,(R,D)) \in \alpha seln \ M \ A$
by ($unfold \ \alpha seln\text{-}def$) $auto$

lemma $\alpha selnD:$ $(S,(R,D)) \in \alpha seln \ M \ A \Longrightarrow S=set \ D \wedge MFSM\text{-}ex \ M \ A \ R \ D$
by ($unfold \ \alpha seln\text{-}def$) $auto$

lemma $\alpha\text{seln}D'$: $(S, C) \in \alpha\text{seln } M A \implies S = \text{set } (\text{snd } C) \wedge \text{MFSM-ex } M A (\text{fst } C)$
 $(\text{snd } C)$ **by** $(\text{cases } C, \text{simp add: } \alpha\text{seln}D)$

lemma αseln -single-valued: *single-valued* $((\alpha\text{seln } M A)^{-1})$
by $(\text{unfold } \alpha\text{seln-def}) (\text{auto intro: single-valuedI})$

theorem (in *MFSM*) seln-refines : $\text{seln-R} \leq_{\alpha\text{seln } M A} (\text{ps-R } M A)$ **proof** (rule refinesI)

show $\text{seln-R } O \alpha\text{seln } M A \subseteq \alpha\text{seln } M A O \text{ps-R } M A$ **proof** $(\text{rule refines-compI})$
fix $a c c'$

assume $ABS: (a, c) \in \alpha\text{seln } M A$ **and** $R: (c, c') \in \text{seln-R}$

then obtain $R D t$ **where** $1: c = (R, D) \wedge c' = (R, t\#D) \wedge \text{sel-next } R D = \text{Some } t$ **by** $(\text{unfold seln-R-alt, blast})$

moreover with ABS **have** $2: a = \text{set } D \wedge \text{MFSM-ex } M A R D$ **by** $(\text{unfold } \alpha\text{seln-def, auto})$

ultimately have $3: (\text{set } D, (\text{set } (t\#D))) \in \text{ps-R } M A$ **using** $\text{MFSM-ex.seln-cons1}$ [of $M A R D$] **by** auto

moreover have $(\text{set } (t\#D), (R, t\#D)) \in \alpha\text{seln } M A$ **proof** $(\text{unfold } \alpha\text{seln-def, auto})$

from 2 **have** $\delta A \subseteq \text{set } D$ **using** MFSM-ex.D-above [of $M A R D$] **by** auto

with 3 **have** $\delta A \subseteq \text{set } (t\#D)$ $\text{set } (t\#D) \subseteq \text{ps-upper } M A$ **using** ps-R-below **by** (fast+)

with 2 **show** $\text{MFSM-ex } M A R (t\#D)$ **by** $(\text{unfold MFSM-ex-alt, simp})$

qed

ultimately show $\exists a'. (a, a') \in \text{ps-R } M A \wedge (a', c') \in \alpha\text{seln } M A$ **using** $1 2$ **by** blast

qed

next

show $\alpha\text{seln } M A \text{ “ Domain } (\text{ps-R } M A) \subseteq \text{Domain seln-R}$

apply $(\text{rule refines-domI})$

apply $(\text{unfold } \alpha\text{seln-def seln-R-alt})$

apply $(\text{unfold Domain-def})$

apply (clarsimp)

apply $(\text{fast dest: MFSM-ex.seln-cons2-rev})$

done

qed

9.4.2 Correctness

We have to show that the next-state selector function’s graph refines seln-R . This is trivial because we defined seln-R to be that graph

lemma sns-refines : $\text{graph sel-next-state} \leq_{Id} \text{seln-R}$ **by** $(\text{unfold seln-R-def}) \text{simp}$

interpretation det-impl : $\text{detRef-impl pss-algo-rec sel-next-state seln-R}$

apply $(\text{rule detRef-impl.intro})$

apply $(\text{simp-all only: detRef-wf-transfer[OF seln-R-wf] sns-refines})$

apply $(\text{unfold sel-next-state-def})$

apply auto


```

apply (case-tac sel-next a b, auto)+
done

```

And then infer correctness of the deterministic algorithm

```

theorem (in MFSM-ex) pss-correct:
  assumes D-init: set  $D = \delta A$ 
  shows lang ( $A \mid \delta := \text{set} (\text{snd} (\text{pss-algo-rec} (R,D))) \mid$ ) = pre-star (rules  $M$ )  $A$ 
proof (rule correct)
  have (set  $D, (R,D)) \in \alpha \text{seln } M A$  using prems MFSM-ex.axioms by (auto intro!:
 $\alpha \text{selnI}$  MFSM-ex.intro)
  moreover have  $((R,D), \text{pss-algo-rec} (R,D)) \in \text{ndet-algo} (\text{seln-}R)$  by (simp add:
det-impl.algo-correct)
  ultimately obtain  $d'$  where  $(d', \text{pss-algo-rec} (R,D)) \in \alpha \text{seln } M A \wedge (\text{set } D, d') \in \text{ndet-algo}$ 
 $(\text{ps-}R \ M \ A)$  using refines-ndet-algo[OF seln-refines] by blast
  moreover hence  $d' = \text{set} (\text{snd} (\text{pss-algo-rec} (R,D)))$  by (blast dest:  $\alpha \text{seln} d'$ )
  ultimately show  $(\delta A, \text{set} (\text{snd} (\text{pss-algo-rec} (R, D)))) \in \text{ndet-algo} (\text{ps-}R \ M \ A)$ 
using D-init by simp
qed

```

```

corollary (in MFSM) pss-correct:
  assumes repr: set  $D = \delta A (R, \text{rules } M) \in \text{rules-repr}$ 
  shows lang ( $A \mid \delta := \text{set} (\text{snd} (\text{pss-algo-rec} (R,D))) \mid$ ) = pre-star (rules  $M$ )  $A$ 
proof –
  interpret MFSM-ex sep  $M \ M \ A \ R \ D$ 
  apply simp-all
  apply (rule MFSM-ex.intro)
  apply (simp-all add: prems)
  apply (rule MFSM-ex.axioms.intro)
  apply (simp-all add: repr initial-delta-below)
  done
  from repr show ?thesis by (simp add: pss-correct)
qed

```

Generate executable code

```

code-module DPN file DPN.sml
  contains pss-algo-rec

```

end

References

- [1] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.