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

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* $= ('c \times 'a \times 'c)$ *set*

Reflexive-transitive closure of LTS

inductive-set *trcl* :: $('c,'a)$ *LTS* $\Rightarrow$ $('c,'a$ *list*$)$ *LTS* **for** t **where**
empty[*simp*]: $(c,[],c) \in$ *trcl* t |
cons[*simp*]: $[\![(c,a,c') \in t; (c',w,c'') \in trcl\ t]\!] \Longrightarrow (c,a\#w,c'') \in trcl\ t$

1.2 Basic properties of reflexive-transitive closure

lemma *trcl-empty-cons*[*simp*]: $(c,[],c')\in trcl\ t \Longrightarrow c=c'$ ⟨*proof*⟩
lemma *trcl-single*: $(c,[a],c') \in trcl\ t \Longrightarrow (c,a,c') \in t$ ⟨*proof*⟩
lemma *trcl-uncons*: $(c,a\#w,c')\in trcl\ t \Longrightarrow \exists\, ch\ .\ (c,a,ch)\in t \wedge (ch,w,c') \in trcl\ t$
⟨*proof*⟩
lemma *trcl-one-elem*: $(c,e,c')\in t \Longrightarrow (c,[e],c')\in trcl\ t$ ⟨*proof*⟩

lemma *trcl-concat*: !! c . $[\![(c,w1,c')\in trcl\ t; (c',w2,c'')\in trcl\ t]\!] \Longrightarrow (c,w1@w2,c'')\in trcl\ t$
⟨*proof*⟩

lemma *trcl-unconcat*: !! c . $(c,w1@w2,c')\in trcl\ t \Longrightarrow \exists\, ch\ .\ (c,w1,ch)\in trcl\ t \wedge (ch,w2,c')\in trcl\ t$ ⟨*proof*⟩

lemma *trcl-mono*: $A \subseteq B \Longrightarrow trcl\ A \subseteq trcl\ B$
⟨*proof*⟩

lemma *trcl-struct*: !!s. $[\![(s,w,s')\in trcl\ D;\ D\subseteq S\times A\times S']\!] \Longrightarrow (s=s' \wedge w=[]) \vee (s\in S \wedge s'\in S' \wedge w\in lists\ A)$ ⟨*proof*⟩

lemma *trcl-structE*: $[\![(s,w,s')\in trcl\ D;\ D\subseteq S\times A\times S';\ [\![s=s';\ w=[]]\!] \Longrightarrow P;\ [\![s\in S;\ s'\in S';\ w\in lists\ A]\!] \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

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

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

translations
$s \hookrightarrow_a s'$ => (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 $tr :: ('c,'l)\ SRS \Rightarrow ('c\ list,\ 'l)\ LTS$ **for** S
where
rewrite: $(s \hookrightarrow_a s') \in S \Longrightarrow (ep@s@es,a,ep@s'@es) \in tr\ S$

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*: $(s,a,s')\in tr\ S \Longrightarrow (wp@s@ws,a,wp@s'@ws)\in tr\ S$ ⟨*proof*⟩

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

corollary *srs-ext-cons*: $(s,w,s')\in trcl\ (tr\ S) \Longrightarrow (e\#s,w,e\#s')\in trcl\ (tr\ S)$ $\langle proof \rangle$
corollary *srs-ext-pre*: $(s,w,s')\in trcl\ (tr\ S) \Longrightarrow (wp@s,w,wp@s')\in trcl\ (tr\ S)$ $\langle proof \rangle$
corollary *srs-ext-post*: $(s,w,s')\in trcl\ (tr\ S) \Longrightarrow (s@ws,w,s'@ws)\in trcl\ (tr\ S)$ $\langle proof \rangle$

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-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-rel* :: $('s\times's)\ set \Rightarrow ('s\times's)\ set$ **for** R
where
A-rel-base: $[\![(s,s')\in R;\ s'\notin Domain\ R]\!] \Longrightarrow (s,s')\in A\text{-}rel\ R$ |
A-rel-step: $[\![(s,sh)\in R;\ (sh,s')\in A\text{-}rel\ R]\!] \Longrightarrow (s,s')\in A\text{-}rel\ R$

3.1 Basic properties

The algorithm just terminates at terminating states

lemma *termstate*: $(s,s')\in A\text{-}rel\ R \Longrightarrow s'\notin Domain\ R$ $\langle proof \rangle$

lemma *dom-subset*: *Domain* (*A-rel R*) ⊆ *Domain R* ⟨*proof*⟩

We can use invariants to reason over properties of the algorithm

constdefs
is-inv R s0 P == *P s0* ∧ (∀ *s s'*. (*s*,*s'*)∈*R* ∧ *P s* ⟶ *P s'*)

lemma *inv*: ⟦(*s0*,*sf*)∈*A-rel R*; *is-inv R s0 P*⟧ ⟹ *P sf* ⟨*proof*⟩
lemma *invI*: ⟦*P s0*; !! *s s'*. ⟦(*s*,*s'*)∈*R*; *P s*⟧ ⟹ *P s'*⟧ ⟹ *is-inv R s0 P* ⟨*proof*⟩
lemma *inv2*: ⟦(*s0*,*sf*)∈*A-rel R*; *P s0*; !! *s s'*. ⟦(*s*,*s'*)∈*R*; *P s*⟧ ⟹ *P s'*⟧ ⟹ *P sf* ⟨*proof*⟩

To establish new invariants, we can use already existing invariants

lemma *inv-useI*: ⟦*P s0*; !! *s s'*. ⟦(*s*,*s'*)∈*R*; *P s*; !!*P'*. *is-inv R s0 P'* ⟹ *P' s* ⟧ ⟹ *P s'* ⟧ ⟹ *is-inv R s0* (λ*s*. *P s* ∧ (∀ *P'*. *is-inv R s0 P'* ⟶ *P' s*))
⟨*proof*⟩

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

lemma *wf-dom-eq*: *wf* (R^{-1}) ⟹ *Domain R* = *Domain* (*A-rel R*) ⟨*proof*⟩

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* ⇒ (*'r*∗*'s*) *set* ⇒ (*'r*∗*'r*) *set* ⇒ *bool* (-≤-- [*50*,*50*,*50*] *50*)
$R \le_\alpha S$ == *R O* α ⊆ α *O S* ∧ α “ *Domain S* ⊆ *Domain R*

lemma *refinesI*: ⟦*R O* α ⊆ α *O S*; α “ *Domain S* ⊆ *Domain R*⟧ ⟹ $R \le_\alpha S$
⟨*proof*⟩
lemma *refinesE*: $R \le_\alpha S$ ⟹ *R O* α ⊆ α *O S*
$R \le_\alpha S$ ⟹ α “ *Domain S* ⊆ *Domain R*
⟨*proof*⟩

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'*. ⟦ (*a*,*c*)∈α; (*c*,*c'*)∈*S* ⟧ ⟹ ∃ *a'*. (*a*,*a'*)∈*R* ∧ (*a'*,*c'*)∈α
shows *S O* α ⊆ α *O R* ⟨*proof*⟩

lemma *refines-compE*: ⟦*S O* α ⊆ α *O R*; (*a*,*c*)∈α; (*c*,*c'*)∈*S*⟧ ⟹ ∃ *a'*. (*a*,*a'*)∈*R* ∧ (*a'*,*c'*)∈α ⟨*proof*⟩

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: !! $a\ a'\ c$. $[\![(a,c)\in\alpha;\ (a,a')\in R\ ]\!] \Longrightarrow c\in \mathit{Domain}\ S$
  shows α `` $\mathit{Domain}\ R \subseteq \mathit{Domain}\ S$ ⟨*proof*⟩

lemma *refines-domE*: $[\![\alpha$ `` $\mathit{Domain}\ R \subseteq \mathit{Domain}\ S;\ (a,c)\in\alpha;\ (a,a')\in R]\!] \Longrightarrow c\in \mathit{Domain}\ S$ ⟨*proof*⟩

lemma *refinesI2*:
  assumes A: !! $a\ c\ c'$. $[\![\ (a,c)\in\alpha;\ (c,c')\in S\ \ ]\!] \Longrightarrow \exists\, a'.\ (a,a')\in R \land (a',c')\in\alpha$
  assumes B: !! $a\ a'\ c$. $[\![(a,c)\in\alpha;\ (a,a')\in R\ ]\!] \Longrightarrow c\in \mathit{Domain}\ S$
  shows $S\leq_\alpha R$ ⟨*proof*⟩

lemma *refinesE2*:
  $[\![S\leq_\alpha R;\ (a,c)\in\alpha;\ (c,c')\in S]\!] \Longrightarrow \exists\, a'.\ (a,a')\in R \land (a',c')\in\alpha$
  $[\![S\leq_\alpha R;\ (a,c)\in\alpha;\ (a,a')\in R]\!] \Longrightarrow c\in \mathit{Domain}\ S$
  ⟨*proof*⟩

Reflexivity of identity refinement

lemma *refines-id-refl*[*intro*!, *simp*]: $R\leq_{Id} R$ ⟨*proof*⟩

Transitivity of refinement

lemma *refines-trans*: **assumes** R: $R\ \leq_\alpha\ S\quad S\ \leq_\beta\ T$ **shows** $R\leq_{\alpha\ O\ \beta}\ T$ ⟨*proof*⟩

Property transfer lemma

lemma *refines-A-rel*[*rule-format*]:
  assumes R: $R\leq_\alpha S$ **and** A: $(r,r')\in \mathit{A\text{-}rel}\ R$
  shows $\forall\, s.\ (s,r)\in\alpha \longrightarrow (\exists\, s'.\ (s',r')\in\alpha \land (s,s')\in \mathit{A\text{-}rel}\ S)$
  ⟨*proof*⟩

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

lemma *refines-A-rel-sv*: $[\![R\leq_\alpha S;\ (r,r')\in \mathit{A\text{-}rel}\ R;\ \mathit{single\text{-}valued}\ (\alpha^{-1});\ (s,r)\in\alpha;\ (s',r')\in\alpha]\!] \Longrightarrow (s,s')\in \mathit{A\text{-}rel}\ S$ ⟨*proof*⟩

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-rel R) ∪ {(s,s) | s. s∉$Domain$ R}

lemma *ndet-algo-A-rel*: ⟦x∈$Domain$ R; (x,y)∈*ndet-algo* R⟧ ⟹ (x,y)∈A-rel R
⟨*proof*⟩

lemma *ndet-algoE*: ⟦(s,s')∈*ndet-algo* R; ⟦(s,s')∈A-rel R⟧ ⟹ P; ⟦ s=s'; s∉$Domain$ R⟧ ⟹ P⟧ ⟹ P ⟨*proof*⟩
lemma *ndet-algoE'*: ⟦(s,s')∈*ndet-algo* R; ⟦(s,s')∈A-rel R; s∈$Domain$ R; s'∉$Domain$ R⟧ ⟹ P; ⟦ s=s'; s∉$Domain$ R⟧ ⟹ P⟧ ⟹ P
⟨*proof*⟩

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}) ⟹ *Domain* (*ndet-algo* R) = *UNIV*
⟨*proof*⟩

The result of the algorithm is always a terminating state

lemma *termstate-ndet-algo*: (s,s')∈*ndet-algo* R ⟹ s'∉$Domain$ R ⟨*proof*⟩

Property transfer lemma for *ndet-algo*

lemma *refines-ndet-algo*[*rule-format*]:
assumes R: $S \leq_\alpha R$ **and** A: (c,c')∈*ndet-algo* S
shows ∀ a. (a,c)∈α ⟶ (∃ a'. (a',c')∈α ∧ (a,a')∈*ndet-algo* R)
⟨*proof*⟩

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

lemma *refines-ndet-algo-sv*: ⟦$S \leq_\alpha R$; (c,c')∈*ndet-algo* S; *single-valued* (α^{-1}); (a,c)∈α; (a',c')∈α⟧ ⟹ (a,a')∈*ndet-algo* R ⟨*proof*⟩

3.4 Well-foundedness

lemma *wf-imp-minimal*: ⟦*wf* S; x∈Q⟧ ⟹ ∃ z∈Q. (∀ x. (x,z)∈S ⟶ x∉Q) ⟨*proof*⟩

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. ⟦ r∈$Domain$ R ⟧ ⟹ (s r,r)∈α r ∧ $R \leq_{\alpha\ r}$ S r ∧ *wf* ((S r)$^{-1}$)
shows *wf* (R^{-1})
⟨*proof*⟩

3.4.1 The relations > and ⊃ on finite domains

constdefs
greaterN N == $\{(i,j)\ .\ j<i\ \&\ i\leq(N::nat)\}$
greaterS S == $\{(a,b)\ .\ b\subset a\ \&\ a\subseteq(S::'a\ set)\}$

> on initial segment of nat is well founded

lemma *wf-greaterN*: *wf* (*greaterN N*)
⟨*proof*⟩

Strict version of *card-mono*

lemma *card-mono-strict*: ⟦*finite B*; $A\subset B$⟧ $\Longrightarrow$ *card A* < *card B* ⟨*proof*⟩

⊃ on finite sets is well founded

This is shown here by embedding the ⊃ relation into the > relation, using cardinality

lemma *wf-greaterS*[*recdef-wf*]: *finite S* $\Longrightarrow$ *wf* (*greaterS S*) ⟨*proof*⟩

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*: !!*r r'*. $(r,r')\in R \Longrightarrow \alpha\ r \subset \alpha\ r' \wedge \alpha\ r' \subseteq U$
assumes *finite*: *finite U*
shows *wf* (R^{-1})
⟨*proof*⟩

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 $(graph\ sel)^{-1}$ is well-founded. If we already know that R^{-1} is well-founded, this property transfers to $(graph\ sel)^{-1}$.

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

algo s = *case sel s of None* ⇒ *s* | *Some s'* ⇒ *algo s'*

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

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*]: $f\,x = Some\ x' \Longrightarrow (x,x') \in graph\ f$ ⟨*proof*⟩
lemma *graphD*[*dest*]: $(x,x') \in graph\ f \Longrightarrow f\,x = Some\ x'$ ⟨*proof*⟩
lemma *graph-dom-iff1*: $(x \notin Domain\ (graph\ f)) = (f\,x = None)$ ⟨*proof*⟩
lemma *graph-dom-iff2*: $(x \in Domain\ (graph\ f)) = (f\,x \neq None)$ ⟨*proof*⟩

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

lemma *detRef-eq*: $(graph\ sel \leq_{Id} R) = ((\forall s\ s'.\ sel\ s = Some\ s' \longrightarrow (s,s') \in R) \wedge (\forall s.\ sel\ s = None \longrightarrow s \notin Domain\ R))$
  ⟨*proof*⟩

lemma *detRef-wf-transfer*: $[\![wf\ (R^{-1});\ graph\ sel \leq_{Id} R\ ]\!] \Longrightarrow wf\ ((graph\ sel)^{-1})$
  ⟨*proof*⟩

3.5.3 Recursive characterization

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

lemma (**in** *detRef-impl*) *sel-cons*:
  $sel\ s = Some\ s' \Longrightarrow (s,s') \in R$
  $sel\ s = None \Longrightarrow s \notin Domain\ R$
  $s \in Domain\ R \Longrightarrow \exists s'.\ sel\ s = Some\ s'$
  $s \notin Domain\ R \Longrightarrow sel\ s = None$
  ⟨*proof*⟩

lemma (**in** *detRef-impl*) *algo-correct*: $(s,algo\ s) \in ndet\text{-}algo\ R$ ⟨*proof*⟩

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*: $(q,l,q')\in\delta\ A \Longrightarrow q\in Q\ A \wedge l\in\Sigma\ A \wedge q'\in Q\ A$ — 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* $(\Sigma\ A)$ — The alphabet is finite

4.2 Basic properties

lemma (**in** *FSM*) *finite-delta-dom*: *finite* $(Q\ A \times \Sigma\ A \times Q\ A)$ ⟨*proof*⟩

lemma (**in** *FSM*) *finite-delta*: *finite* $(\delta\ A)$ ⟨*proof*⟩

4.3 Reflexive, transitive closure of transition relation

Reflexive transitive closure on restricted domain

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

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

lemma *trclAD-empty-cons*[*simp*]: $(c,[],c')\in$ *trclAD* $A\ D \Longrightarrow c=c'$ ⟨*proof*⟩
lemma *trclAD-single*: $(c,[a],c') \in$ *trclAD* $A\ D \Longrightarrow (c,a,c') \in D$ ⟨*proof*⟩
lemma *trclAD-elems*: $(c,w,c')\in$ *trclAD* $A\ D \Longrightarrow c\in Q\ A \wedge w\in$ *lists* $(\Sigma\ A) \wedge c'\in Q\ A$ ⟨*proof*⟩

lemma *trclAD-one-elem*: $[\![c{\in}Q\ A;\ e{\in}\Sigma\ A;\ c'{\in}Q\ A;\ (c,e,c'){\in}D]\!] \Longrightarrow (c,[e],c'){\in}trclAD\ A\ D$ ⟨*proof*⟩

lemma *trclAD-uncons*: $(c,a\#w,c'){\in}trclAD\ A\ D \Longrightarrow \exists\ ch\ .\ (c,a,ch){\in}D \wedge (ch,w,c') \in trclAD\ A\ D \wedge c{\in}Q\ A \wedge a{\in}\Sigma\ A$
⟨*proof*⟩

lemma *trclAD-concat*: !! c . $[\![\ (c,w1,c'){\in}trclAD\ A\ D;\ (c',w2,c''){\in}trclAD\ A\ D\ ]\!] \Longrightarrow (c,w1@w2,c''){\in}trclAD\ A\ D$
⟨*proof*⟩

lemma *trclAD-unconcat*: !! c . $(c,w1@w2,c'){\in}trclAD\ A\ D \Longrightarrow \exists\ ch\ .\ (c,w1,ch){\in}trclAD\ A\ D \wedge (ch,w2,c'){\in}trclAD\ A\ D$ ⟨*proof*⟩

lemma *trclAD-eq*: $[\![Q\ A = Q\ A';\ \Sigma\ A = \Sigma\ A']\!] \Longrightarrow trclAD\ A\ D = trclAD\ A'\ D$
⟨*proof*⟩

lemma *trclAD-mono*: $D{\subseteq}D' \Longrightarrow trclAD\ A\ D \subseteq trclAD\ A\ D'$
⟨*proof*⟩

lemma *trclAD-mono-adv*: $[\![D{\subseteq}D';\ Q\ A = Q\ A';\ \Sigma\ A = \Sigma\ A']\!] \Longrightarrow trclAD\ A\ D \subseteq trclAD\ A'\ D'$ ⟨*proof*⟩

4.3.1 Relation of *trclAD* and *trcl*

lemma *trclAD-by-trcl1*: $trclAD\ A\ D \subseteq (trcl\ (D \cap (Q\ A \times \Sigma\ A \times Q\ A)) \cap (Q\ A \times lists\ (\Sigma\ A) \times Q\ A))$
⟨*proof*⟩

lemma *trclAD-by-trcl2*: $(trcl\ (D \cap (Q\ A \times \Sigma\ A \times Q\ A)) \cap (Q\ A \times lists\ (\Sigma\ A) \times Q\ A)) \subseteq trclAD\ A\ D$ ⟨*proof*⟩

lemma *trclAD-by-trcl*: $trclAD\ A\ D = (trcl\ (D \cap (Q\ A \times \Sigma\ A \times Q\ A)) \cap (Q\ A \times lists\ (\Sigma\ A) \times Q\ A))$
⟨*proof*⟩

lemma *trclAD-by-trcl'*: $trclAD\ A\ D = (trcl\ (D \cap (Q\ A \times \Sigma\ A \times Q\ A)) \cap (Q\ A \times UNIV \times UNIV))$
⟨*proof*⟩

lemma *trclAD-by-trcl''*: $[\![\ D{\subseteq}Q\ A \times \Sigma\ A \times Q\ A\ ]\!] \Longrightarrow trclAD\ A\ D = trcl\ D \cap (Q\ A \times UNIV \times UNIV)$
⟨*proof*⟩

lemma *trclAD-subset-trcl*: $trclAD\ A\ D \subseteq trcl\ (D)$ ⟨*proof*⟩

4.4 Language of a FSM

constdefs

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

lemma *langs-alt-def*: (*w*$\in$*langs A s*) == ($\exists$*f* . *f*$\in$*F A* & (*s*,*w*,*f*) $\in$ *trclA A*) ⟨*proof*⟩

4.5 Example: Product automaton

constdefs

prod-fsm A1 A2 == ⦇ *Q*=*Q A1* × *Q A2*, Σ=Σ *A1* ∩ Σ *A2*, *δ* = { ((*s*,*t*),*a*,(*s'*,*t'*)) . (*s*,*a*,*s'*)$\in$*δ A1* ∧ (*t*,*a*,*t'*)$\in$*δ A2* }, *s0*=(*s0 A1*,*s0 A2*), *F* = {(*s*,*t*) . *s*$\in$*F A1* ∧ *t*$\in$*F A2*} ⦈

lemma *prod-inter-1*: !! *s s' f f'* . ((*s*,*s'*),*w*,(*f*,*f'*)) $\in$ *trclA* (*prod-fsm A A'*) ⟹ (*s*,*w*,*f*) $\in$ *trclA A* ∧ (*s'*,*w*,*f'*) $\in$ *trclA A'* ⟨*proof*⟩

lemma *prod-inter-2*: !! *s s' f f'* . (*s*,*w*,*f*) $\in$ *trclA A* ∧ (*s'*,*w*,*f'*) $\in$ *trclA A'* ⟹ ((*s*,*s'*),*w*,(*f*,*f'*)) $\in$ *trclA* (*prod-fsm A A'*) ⟨*proof*⟩

lemma *prod-F*: (*a*,*b*)$\in$*F* (*prod-fsm A B*) = (*a*$\in$*F A* ∧ *b*$\in$*F B*) ⟨*proof*⟩
lemma *prod-FI*: ⟦*a*$\in$*F A*; *b*$\in$*F B*⟧ ⟹ (*a*,*b*)$\in$*F* (*prod-fsm A B*) ⟨*proof*⟩

lemma *prod-fsm-langs*: *langs* (*prod-fsm A B*) (*s*,*t*) = *langs A s* ∩ *langs B t*
⟨*proof*⟩

lemma *prod-FSM-intro*: *FSM A1* ⟹ *FSM A2* ⟹ *FSM* (*prod-fsm A1 A2*) ⟨*proof*⟩

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 symbol[1], 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 $[p,\gamma] \hookrightarrow_a p1\#w1@\sharp\#p2\#w2$ where p,$p1$,$p2$ are control symbols, $w1$,$w2$ are sequences of stack symbols, a is a label and $\sharp$ is the separator.

locale *DPN* =
  fixes *M*
  fixes *separator* ($\sharp$)
  defines *sep-def*: $\sharp$ == *sep M*
  assumes *sym-finite*: *finite* (*csyms M*) *finite* (*ssyms M*)
  assumes *sym-disjoint*: *csyms M* $\cap$ *ssyms M* = {} $\sharp \notin$ *csyms M* $\cup$ *ssyms M*
  assumes *lab-finite*: *finite* (*labels M*)
  assumes *rules-finite*: *finite* (*rules M*)
  assumes *rule-fmt*: $r \in$ *rules M* $\Longrightarrow$
    ($\exists$ p γ 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 γ 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=$p\#[\gamma] \hookrightarrow_a p1\#w1@\sharp\#p2\#w2$)

lemma (**in** *DPN*) *sep-fold*: *sep M* == $\sharp$ ⟨*proof*⟩

lemma (**in** *DPN*) *sym-disjoint'*: *sep M* $\notin$ *csyms M* $\cup$ *ssyms M* ⟨*proof*⟩

5.1.2 Basic properties

lemma (**in** *DPN*) *syms-part*: $x\in$*csyms M* $\Longrightarrow$ $x\notin$*ssyms M* $x\in$*ssyms M* $\Longrightarrow$ $x\notin$*csyms M* ⟨*proof*⟩
lemma (**in** *DPN*) *syms-sep*: $\sharp\notin$*csyms M* $\sharp\notin$*ssyms M* ⟨*proof*⟩
lemma (**in** *DPN*) *syms-sep'*: *sep M*$\notin$*csyms M* *sep M*$\notin$*ssyms M* ⟨*proof*⟩

lemma (**in** *DPN*) *rule-cases*:
  assumes *A*: $r\in$*rules M*

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

assumes *NOSPAWN*: !! $p\ \gamma\ a\ p'\ w$. ⟦$p{\in}csyms\ M$; $\gamma{\in}ssyms\ M$; $p'{\in}csyms\ M$; $w{\in}lists\ (ssyms\ M)$; $a{\in}labels\ M$; $r{=}p\#[\gamma] \hookrightarrow_a p'\#w$⟧ $\Longrightarrow P$
assumes *SPAWN*: !! $p\ \gamma\ a\ p1\ w1\ p2\ w2$. ⟦$p{\in}csyms\ M$; $\gamma{\in}ssyms\ M$; $p1{\in}csyms\ M$; $w1{\in}lists\ (ssyms\ M)$; $p2{\in}csyms\ M$; $w2{\in}lists\ (ssyms\ M)$; $a{\in}labels\ M$; $r{=}p\#[\gamma] \hookrightarrow_a p1\#w1@\sharp\#p2\#w2$⟧ $\Longrightarrow P$
shows P
⟨*proof*⟩

lemma (**in** *DPN*) *rule-cases'*:
⟦$r{\in}rules\ M$;
!! $p\ \gamma\ a\ p'\ w$. ⟦$p{\in}csyms\ M$; $\gamma{\in}ssyms\ M$; $p'{\in}csyms\ M$; $w{\in}lists\ (ssyms\ M)$; $a{\in}labels\ M$; $r{=}p\#[\gamma] \hookrightarrow_a p'\#w$⟧ $\Longrightarrow P$;
!! $p\ \gamma\ a\ p1\ w1\ p2\ w2$. ⟦$p{\in}csyms\ M$; $\gamma{\in}ssyms\ M$; $p1{\in}csyms\ M$; $w1{\in}lists\ (ssyms\ M)$; $p2{\in}csyms\ M$; $w2{\in}lists\ (ssyms\ M)$; $a{\in}labels\ M$; $r{=}p\#[\gamma] \hookrightarrow_a p1\#w1@(sep\ M)\#p2\#w2$⟧ $\Longrightarrow P$⟧
$\Longrightarrow P$ ⟨*proof*⟩

lemma (**in** *DPN*) *rule-prem-fmt*: $r{\in}rules\ M \Longrightarrow \exists\ p\ \gamma\ a\ c'.\ p{\in}csyms\ M \wedge \gamma{\in}ssyms\ M \wedge a{\in}labels\ M \wedge set\ c' \subseteq csyms\ M \cup ssyms\ M \cup \{\sharp\} \wedge r{=}(p\#[\gamma] \hookrightarrow_a c')$
⟨*proof*⟩

lemma (**in** *DPN*) *rule-prem-fmt'*: $r{\in}rules\ M \Longrightarrow \exists\ p\ \gamma\ a\ c'.\ p{\in}csyms\ M \wedge \gamma{\in}ssyms\ M \wedge a{\in}labels\ M \wedge set\ c' \subseteq csyms\ M \cup ssyms\ M \cup \{sep\ M\} \wedge r{=}(p\#[\gamma] \hookrightarrow_a c')$ ⟨*proof*⟩

lemma (**in** *DPN*) *rule-prem-fmt2*: $[p,\gamma]{\hookrightarrow_a}\ c' \in rules\ M \Longrightarrow p{\in}csyms\ M \wedge \gamma{\in}ssyms\ M \wedge a{\in}labels\ M \wedge set\ c' \subseteq csyms\ M \cup ssyms\ M \cup \{\sharp\}$ ⟨*proof*⟩
lemma (**in** *DPN*) *rule-prem-fmt2'*: $[p,\gamma]{\hookrightarrow_a}\ c' \in rules\ M \Longrightarrow p{\in}csyms\ M \wedge \gamma{\in}ssyms\ M \wedge a{\in}labels\ M \wedge set\ c' \subseteq csyms\ M \cup ssyms\ M \cup \{sep\ M\}$ ⟨*proof*⟩

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

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 partioned 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*: $\Sigma\ A = csyms\ M \cup ssyms\ M \cup \{\sharp\}$
  assumes *states-part*: $sstates\ A \cap cstates\ A = \{\}$ $Q\ A = sstates\ A \cup cstates\ A$
  assumes *uniqueSp*: $[\![s\in cstates\ A;\ p\in csyms\ M]\!] \Longrightarrow sp\ A\ s\ p \in sstates\ A$ $[\![p\in csyms\ M;\ p'\in csyms\ M;\ s\in cstates\ A;\ s'\in cstates\ A;\ sp\ A\ s\ p = sp\ A\ s'\ p']\!] \Longrightarrow s=s' \wedge p=p'$
  assumes *delta-fmt*: $\delta\ A \subseteq (sstates\ A \times ssyms\ M \times (sstates\ A - \{sp\ A\ s\ p \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\})) \cup (sstates\ A \times \{\sharp\} \times cstates\ A) \cup \{(s,p,sp\ A\ s\ p) \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\}$
  $\delta\ A \supseteq \{(s,p,sp\ A\ s\ p) \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\}$
  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'*: $\Sigma\ A = csyms\ M \cup ssyms\ M \cup \{sep\ M\}$ ⟨*proof*⟩
lemma (**in** *MFSM*) *delta-fmt'*: $\delta\ A \subseteq (sstates\ A \times ssyms\ M \times (sstates\ A - \{sp\ A\ s\ p \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\})) \cup (sstates\ A \times \{sep\ M\} \times cstates\ A) \cup \{(s,p,sp\ A\ s\ p) \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\}$
  $\delta\ A \supseteq \{(s,p,sp\ A\ s\ p) \mid s\ p\ .\ s\in cstates\ A \wedge p\in csyms\ M\}$ ⟨*proof*⟩

5.2.2 Basic properties

lemma (**in** *MFSM*) *finite-cs-states*: *finite* (*sstates A*) *finite* (*cstates A*)
⟨*proof*⟩

lemma (**in** *MFSM*) *sep-out-syms*: $x\in csyms\ M \Longrightarrow x \neq \sharp$ $x\in ssyms\ M \Longrightarrow x \neq \sharp$
⟨*proof*⟩
lemma (**in** *MFSM*) *sepI*: $[\![x\in\Sigma\ A; x\notin csyms\ M;\ x\notin ssyms\ M]\!] \Longrightarrow x=\sharp$ ⟨*proof*⟩
lemma (**in** *MFSM*) *sep-out-syms'*: $x\in csyms\ M \Longrightarrow x \neq sep\ M$ $x\in ssyms\ M \Longrightarrow x \neq sep\ M$ ⟨*proof*⟩
lemma (**in** *MFSM*) *sepI'*: $[\![x\in\Sigma\ A; x\notin csyms\ M;\ x\notin ssyms\ M]\!] \Longrightarrow x=sep\ M$ ⟨*proof*⟩

lemma (**in** *MFSM*) *states-partI1*: $x\in sstates\ A \Longrightarrow \neg x\in cstates\ A$ ⟨*proof*⟩
lemma (**in** *MFSM*) *states-partI2*: $x\in cstates\ A \Longrightarrow \neg x\in sstates\ A$ ⟨*proof*⟩

lemma (**in** *MFSM*) *states-part-elim*[*elim*]: $[\![q{\in}Q\ A;\ q{\in}sstates\ A \Longrightarrow P;\ q{\in}cstates\ A \Longrightarrow P]\!] \Longrightarrow P$ ⟨*proof*⟩

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*: $[\![(q,p,q'){\in}\delta\ A;\ q{\in}sstates\ A \wedge p{\in}ssyms\ M \wedge q'{\in}sstates\ A \wedge q'{\notin}\{sp\ A\ s\ p \mid s\ p\ .\ s{\in}cstates\ A \wedge p{\in}csyms\ M\} \Longrightarrow P;$
$q{\in}sstates\ A \wedge p{=}\sharp \wedge q'{\in}cstates\ A \Longrightarrow P;$
$q{\in}cstates\ A \wedge p{\in}csyms\ M \wedge q'{=}sp\ A\ q\ p \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma (**in** *MFSM*) *delta-elems*: $(q,p,q'){\in}\delta\ A \Longrightarrow q{\in}sstates\ A \wedge ((p{\in}ssyms\ M \wedge q'{\in}sstates\ A \wedge (q'{\notin}\{sp\ A\ s\ p \mid s\ p\ .\ s{\in}cstates\ A \wedge p{\in}csyms\ M\})) \vee (p{=}\sharp \wedge q'{\in}cstates\ A)) \vee (q{\in}cstates\ A \wedge p{\in}csyms\ M \wedge q'{=}sp\ A\ q\ p)$
⟨*proof*⟩

lemma (**in** *MFSM*) *delta-cases′*: $[\![(q,p,q'){\in}\delta\ A;\ q{\in}sstates\ A \wedge p{\in}ssyms\ M \wedge q'{\in}sstates\ A \wedge q'{\notin}\{sp\ A\ s\ p \mid s\ p\ .\ s{\in}cstates\ A \wedge p{\in}csyms\ M\} \Longrightarrow P;$
$q{\in}sstates\ A \wedge p{=}sep\ M \wedge q'{\in}cstates\ A \Longrightarrow P;$
$q{\in}cstates\ A \wedge p{\in}csyms\ M \wedge q'{=}sp\ A\ q\ p \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma (**in** *MFSM*) *delta-elems′*: $(q,p,q'){\in}\delta\ A \Longrightarrow q{\in}sstates\ A \wedge ((p{\in}ssyms\ M \wedge q'{\in}sstates\ A \wedge (q'{\notin}\{sp\ A\ s\ p \mid s\ p\ .\ s{\in}cstates\ A \wedge p{\in}csyms\ M\})) \vee (p{=}sep\ M \wedge q'{\in}cstates\ A)) \vee (q{\in}cstates\ A \wedge p{\in}csyms\ M \wedge q'{=}sp\ A\ q\ p)$
⟨*proof*⟩

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*: $[\![p{\in}csyms\ M;\ s{\in}cstates\ A]\!] \Longrightarrow (s,p,sp\ A\ s\ p) \in \delta\ A$
⟨*proof*⟩

lemma (**in** *MFSM*) *cstate-succ-ex′*: $[\![p{\in}csyms\ M;\ s{\in}cstates\ A;\ \delta\ A \subseteq D]\!] \Longrightarrow (s,p,sp\ A\ s\ p) \in D$ ⟨*proof*⟩

lemma (**in** *MFSM*) *cstate-succ-unique*: $[\![s{\in}cstates\ A;\ (s,p,x){\in}\delta\ A]\!] \Longrightarrow p{\in}csyms\ M \wedge x{=}sp\ A\ s\ p$ ⟨*proof*⟩

Transitions labeled with control symbols only leave from control states

lemma (**in** *MFSM*) *csym-from-cstate*: $[\![(s,p,s'){\in}\delta\ A;\ p{\in}csyms\ M]\!] \Longrightarrow s{\in}cstates$

A ⟨*proof*⟩

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

lemma (**in** *MFSM*) *sp-pred-ex*: ⟦*s*∈*cstates A*; *p*∈*csyms M*⟧ ⟹ (*s*,*p*,*sp A s p*)∈δ *A* ⟨*proof*⟩
lemma (**in** *MFSM*) *sp-pred-unique*: ⟦*s*∈*cstates A*; *p*∈*csyms M*; (*s'*,*p'*,*sp A s p*)∈δ *A*⟧ ⟹ *s'*=*s* ∧ *p'*=*p* ∧ *s'*∈*cstates A* ∧ *p'*∈*csyms M* ⟨*proof*⟩

Only separators lead from stack states to control states

lemma (**in** *MFSM*) *sep-in-between*: ⟦*s*∈*sstates A*; *s'*∈*cstates A*; (*s*,*p*,*s'*)∈δ *A*⟧ ⟹ *p*=♯ ⟨*proof*⟩
lemma (**in** *MFSM*) *sep-to-cstate*: ⟦(*s*,♯,*s'*)∈δ *A*⟧ ⟹ *s*∈*sstates A* ∧ *s'*∈*cstates A* ⟨*proof*⟩

Stack states do not have successors labelled with control symbols

lemma (**in** *MFSM*) *sstate-succ*: ⟦*s*∈*sstates A*; (*s*,γ,*s'*)∈δ *A*⟧ ⟹ γ ∉ *csyms M* ⟨*proof*⟩
lemma (**in** *MFSM*) *sstate-succ2*: ⟦*s*∈*sstates A*; (*s*,γ,*s'*)∈δ *A*; γ≠♯⟧ ⟹ γ∈*ssyms M* ∧ *s'*∈*sstates A* ⟨*proof*⟩

M-automata do not accept the empty word

lemma (**in** *MFSM*) *not-empty*[*iff*]: []∉*lang A*
⟨*proof*⟩

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*: !!*s* . ⟦*s*∈*sstates A*; (*s*,*w*,*f*)∈*trclA A*⟧ ⟹ (*f*∈*sstates A* ∧ *w*∈*lists* (*ssyms M*)) ∨ (∃ *w1 w2 t*. *w*=*w1*@♯#*w2* ∧ *w1*∈*lists* (*ssyms M*) ∧ *t*∈*sstates A* ∧ (*s*,*w1*,*t*)∈*trclA A* ∧ (*t*,♯#*w2*,*f*)∈*trclA A*)
⟨*proof*⟩

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*∈*cstates A* (*s*,*c*,*f*)∈*trclA A* *f*∈*sstates A*
assumes *SINGLE*: !! *p w* . ⟦*c*=*p*#*w*; *p*∈*csyms M*; *w*∈*lists* (*ssyms M*); (*s*,*p*,*sp A s p*)∈δ *A*; (*sp A s p*,*w*,*f*)∈*trclA A*⟧ ⟹ *P*
assumes *CONC*: !! *p w cr t s'* . ⟦*c*=*p*#*w*@♯#*cr*; *p*∈*csyms M*; *w*∈*lists* (*ssyms M*); *t*∈*sstates A*; *s'*∈*cstates A*; (*s*,*p*,*sp A s p*)∈δ *A*; (*sp A s p*,*w*,*t*)∈*trclA A*; (*t*,♯,*s'*)∈δ *A*; (*s'*,*cr*,*f*)∈*trclA A*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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-star* :: $('c,'l)$ *SRS* $\Rightarrow$ $('s,'c,'e)$ *FSM-rec-scheme* $\Rightarrow$ $'c$ *list set* (pre^*)
for Δ L
where
pre-refl: $c \in lang\ L \Longrightarrow c \in pre^*\ \Delta\ L$ |
pre-step: $[\![c' \in pre^*\ \Delta\ L;\ (c,a,c') \in tr\ \Delta]\!] \Longrightarrow c \in pre^*\ \Delta\ L$

Alternative characterization of $pre^*\ \Delta\ L$

lemma *pre-star-alt*: $pre^*\ \Delta\ L == \{c\ .\ \exists\, c' \in lang\ L\ .\ \exists\, as\ .\ (c,as,c') \in trcl\ (tr\ \Delta)\}$
⟨*proof*⟩

lemma *pre-star-altI*: $[\![c' \in lang\ L;\ c \hookrightarrow_{as}\ c' \in trcl\ (tr\ \Delta)]\!] \Longrightarrow c \in pre^*\ \Delta\ L$ ⟨*proof*⟩
lemma *pre-star-altE*: $[\![c \in pre^*\ \Delta\ L;\ !!c'\ as.\ [\![c' \in lang\ L;\ c \hookrightarrow_{as}\ c' \in trcl\ (tr\ \Delta)]\!] \Longrightarrow P]\!] \Longrightarrow P$ ⟨*proof*⟩

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. $(q,\ c',\ q') \in trclAD\ A\ D$) 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 $c_1\ @\ c'\ @\ c_2 \in pre^*\ \Delta\ L$, and a rule $(p\ \#\ \gamma,\ a,\ c') \in \Delta$, then we also have $c_1\ @\ p\ \#\ \gamma\ @\ c_2 \in pre^*\ \Delta\ L$. 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-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-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-upper :: ('c,'l,'e1) *DPN-rec-scheme* ⇒ ('s,'c,'e2) *MFSM-rec-scheme* ⇒ ('s,'c) *LTS*

inductive-set *ps-R* :: ('c,'l,'e1) *DPN-rec-scheme* ⇒ ('s,'c,'e2) *MFSM-rec-scheme* ⇒ (('s,'c) *LTS* ∗ ('s,'c) *LTS*) *set* **for** *M A*
where
⟦[p,γ]↪$_a$ c' ∈ *rules M*; (q,c',q')∈*trclAD A D*; (*sp A q p*,γ,q')∉D; D⊆*ps-upper M A*⟧ ⟹ (D,*insert* (*sp A q p*,γ,q') D)∈*ps-R M A*

lemma *ps-R-dom-below*: (D,D')∈*ps-R M A* ⟹ D⊆*ps-upper M A* ⟨*proof*⟩

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-upper M A* as a finite set, and show that the initial transition relation δ *A* of an M-automata is below *ps-upper M A*, and that *ps-R M A* preserves the property of being below the finite set *ps-upper M A*.

In the following, we also establich some properties of transition relations below *ps-upper M A*, that will be used later in the correctness proof.

Note that we use the more fine-grained *ps-upper M A* as upper bound for the termination proof rather than *Q A* × Σ *A* × *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-upper M A* to guarantee that *sp* is only applied to arguments it is specified for. Anyway, the fine-grained *ps-upper M A* bound is also needed for the correctness proof.

defs
ps-upper-def: *ps-upper M A* == (*sstates A* × *ssyms M* × *sstates A*) ∪ (*sstates A* × {*sep M*} × *cstates A*) ∪ {(s,p,*sp A s p*) | s p . s∈*cstates A* ∧ p∈*csyms M*}

lemma (**in** *MFSM*) *ps-upper-cases*: ⟦(s,e,s')∈*ps-upper M A*;
⟦s∈*sstates A*; e∈*ssyms M*; s'∈*sstates A*⟧ ⟹ P;
⟦s∈*sstates A*; e=♯; s'∈*cstates A*⟧ ⟹ P;
⟦s∈*cstates A*; e∈*csyms M*; s'=*sp A s e*⟧ ⟹ P
⟧ ⟹ P
⟨*proof*⟩

lemma (**in** *MFSM*) *ps-upper-cases'*: ⟦(s,e,s')∈*ps-upper M A*;
⟦s∈*sstates A*; e∈*ssyms M*; s'∈*sstates A*⟧ ⟹ P;
⟦s∈*sstates A*; e=*sep M*; s'∈*cstates A*⟧ ⟹ P;
⟦s∈*cstates A*; e∈*csyms M*; s'=*sp A s e*⟧ ⟹ P
⟧ ⟹ P

⟨*proof*⟩

lemma (**in** *MFSM*) *ps-upper-below-trivial*: $\textit{ps-upper}\ M\ A \subseteq Q\ A \times \Sigma\ A \times Q\ A$ ⟨*proof*⟩

lemma (**in** *MFSM*) *ps-upper-finite*: *finite* (*ps-upper M A*) ⟨*proof*⟩

The initial transition relation of the M-automaton is below *ps-upper M A*

lemma (**in** *MFSM*) *initial-delta-below*: $\delta\ A \subseteq \textit{ps-upper}\ M\ A$ ⟨*proof*⟩

Some lemmas about structure of transition relations below *ps-upper M A*

lemma (**in** *MFSM*) *cstate-succ-unique'*: $[\![s{\in}\textit{cstates}\ A;\ (s,p,x){\in}D;\ D{\subseteq}\textit{ps-upper}\ M\ A]\!] \Longrightarrow p{\in}\textit{csyms}\ M \wedge x{=}\textit{sp}\ A\ s\ p$ ⟨*proof*⟩
lemma (**in** *MFSM*) *csym-from-cstate'*: $[\![(s,p,s'){\in}D;\ D{\subseteq}\textit{ps-upper}\ M\ A;\ p{\in}\textit{csyms}\ M]\!] \Longrightarrow s{\in}\textit{cstates}\ A$ ⟨*proof*⟩

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

lemma (**in** *MFSM*) *ctrl-after-sep*: **assumes** *BELOW*: $D \subseteq \textit{ps-upper}\ M\ A$
  assumes *A*: $(q,c',q'){\in}\textit{trclAD}\ A\ D \quad c'{\neq}[]$
  shows $q'{\in}\textit{cstates}\ A = (\textit{last}\ c' = \sharp)$
⟨*proof*⟩

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*: $D \subseteq \textit{ps-upper}\ M\ A$
  assumes *A*: $([p,\gamma],a,c'){\in}\textit{rules}\ M\ q{\in}\textit{cstates}\ A\ (q,c',q'){\in}\textit{trclAD}\ A\ D$
  shows $q'{\in}\textit{sstates}\ A$
⟨*proof*⟩

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*: $(D,D'){\in}\textit{ps-R}\ M\ A$
  shows $D{\subset}D' \wedge D' \subseteq \textit{ps-upper}\ M\ A$
⟨*proof*⟩

As a result of this section, we get the well-foundedness of *ps-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-upper M A*

theorem (**in** *MFSM*) *ps-R-wf*: $\textit{wf}\ ((\textit{ps-R}\ M\ A)^{-1})$ ⟨*proof*⟩

theorem (**in** *MFSM*) *ps-R-above-inv*: $\textit{is-inv}\ (\textit{ps-R}\ M\ A)\ (\delta\ A)\ (\lambda D.\ \delta\ A \subseteq D)$ ⟨*proof*⟩

theorem (**in** *MFSM*) *ps-R-below-inv*: $\textit{is-inv}\ (\textit{ps-R}\ M\ A)\ (\delta\ A)\ (\lambda D.\ D{\subseteq}\textit{ps-upper}\ M\ A)$ ⟨*proof*⟩

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

theorem (**in** *MFSM*) *total*: $\exists\, D.\ (\delta\ A,\ D){\in}\textit{ndet-algo}(\textit{ps-R}\ M\ A)$ ⟨*proof*⟩

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 $c \hookrightarrow_a c'$ that also c is in the saturated language.

theorem (**in** *MFSM*) *sound*: $[\![c \in pre\text{-}star\ (rules\ M)\ A;\ (\delta\ A, s') \in ndet\text{-}algo\ (ps\text{-}R\ M\ A)]\!] \Longrightarrow c \in lang\ (A(\!|\ \delta := s'\ |\!))$
⟨*proof*⟩

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 occurences 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 $\delta\ A$, 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* . $(qs,w,qf) \in trclAD\ A\ D \Longrightarrow P\ w$
assumes *STEP-CASE*: !! *wp ws* . $[\![(qs,wp,s) \in trclAD\ A\ D;\ (s',ws,qf) \in trclAD\ A\ (insert\ (s,a,s')\ D);$!! *wh* . $(qs,wh,s') \in trclAD\ A\ D \Longrightarrow P\ (wh@ws)]\!] \Longrightarrow P\ (wp@a\#ws)$
shows *P w*
⟨*proof*⟩

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-elims-adv*:
$[\![(D,D')\in ps\text{-}R\ M\ A;\ !!\gamma\ s'\ a\ p'\ c'\ p\ q.\ [\![$
$D'=insert\ (sp\ A\ q\ p,\gamma,s')\ D;\ (sp\ A\ q\ p,\gamma,s')\notin D;\ [p,\gamma]\hookrightarrow_a\ p'\#c' \in\ rules\ M;\ (q,p'\#c',s')\in trclAD\ A\ D;$
$p\in csyms\ M;\ \gamma\in ssyms\ M;\ q\in cstates\ A;\ p'\in csyms\ M;\ a\in labels\ M;\ (q,p',sp\ A\ q\ p')\in D;\ (sp\ A\ q\ p',c',s')\in trclAD\ A\ D$
$]\!]\Longrightarrow P\ ]\!]$
$\Longrightarrow P$
⟨*proof*⟩

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 $is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ (\lambda D.$
$(\forall\ w\ .\ \forall p\in csyms\ M.\ \forall q\in cstates\ A.\ (s0\ A,w,sp\ A\ q\ p)\in trclAD\ A\ D \longrightarrow (\exists\ ws\ as.\ (s0\ A,ws,q)\in trclA\ A \wedge (w,as,ws@[p])\in trcl\ (tr\ (rules\ M)))) \wedge$
$(\forall P'.\ is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ P' \longrightarrow P'\ D))$
— We show the thesis by proving that it is an invariant of the saturation procedure

⟨*proof*⟩

Helper lemma to clarify some subgoal in the precision proof:

lemma *trclAD-delta-update-inv*: $trclAD\ (A(\!|\delta:=X|\!))\ D = trclAD\ A\ D$ ⟨*proof*⟩

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

theorem (**in** *MFSM*) *precise-inv*:
shows $is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ (\lambda D.\ (lang\ (A(\!|\delta:=D|\!)) \subseteq pre^*\ (rules\ M)\ A) \wedge (\forall P'.\ is\text{-}inv\ (ps\text{-}R\ M\ A)\ (\delta\ A)\ P' \longrightarrow P'\ D))$
⟨*proof*⟩

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*: $[\![(\delta\ A,D)\in ndet\text{-}algo\ (ps\text{-}R\ M\ A);\ x\in lang\ (A(\!|\ \delta:=D\ |\!))]\!] \Longrightarrow x\in pre\text{-}star\ (rules\ M)\ A$
⟨*proof*⟩

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

theorem (**in** *MFSM*) *correct*: $[\![(\delta\ A,D)\in ndet\text{-}algo\ (ps\text{-}R\ M\ A)]\!] \Longrightarrow lang\ (A(\!|\ \delta{:=}D\ |\!)) = pre\text{-}star\ (rules\ M)\ A\ \langle proof\rangle$

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

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

and returns the correct result

$[\![MFSM\ ?M\ ?A;\ (\delta\ ?A,\ ?D) \in ndet\text{-}algo\ (ps\text{-}R\ ?M\ ?A)]\!] \Longrightarrow lang\ (?A(\!|\delta := ?D|\!)) = pre^*\ (rules\ ?M)\ ?A$

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 wether 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* ⇒ ('s,'c,'m2) *MFSM-rec-scheme* ⇒ ('s,'c) *LTS* ⇒ ('s*'c*'s) ⇒ *bool*
pss-isNext M A D t == *t*∉*D* ∧ (∃ *q p* γ *q' a c'*. *t*=(*sp A q p*,γ,*q'*) ∧ [*p*,γ]$\hookrightarrow_a$ *c'* ∈ *rules M* ∧ (*q*,*c'*,*q'*)∈*trclAD A D*)
pss-next M A D == *if* (∃ *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*) ⇒ *if MFSM M A* ∧ *D*⊆*ps-upper M A then* (*case pss-next M A D of None* ⇒ *None* | *Some t* ⇒ *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* ⟹ *t*∉*D* ∧ (∃ *q p* γ *q' a c'*. *t*=(*sp A q p*,γ,*q'*) ∧ [*p*,γ]$\hookrightarrow_a$ *c'* ∈ *rules M* ∧ (*q*,*c'*,*q'*)∈*trclAD A D*)
⟨*proof*⟩

lemma *pss-nextE2*: *pss-next M A D* = *None* ⟹ ¬(∃ *q p* γ *q' a c' t*. *t*∉*D* ∧ *t*=(*sp A q p*,γ,*q'*) ∧ [*p*,γ]$\hookrightarrow_a$ *c'* ∈ *rules M* ∧ (*q*,*c'*,*q'*)∈*trclAD A D*)
⟨*proof*⟩

lemmas (**in** *MFSM*) *pss-nextE* = *pss-nextE1 pss-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-R* == {(((*M*,*A*),*D*),((*M*,*A*),*insert t D*)) | *M A D t*. *MFSM M A* ∧ *D*⊆*ps-upper M A* ∧ *pss-next M A D* = *Some t*}
⟨*proof*⟩

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 $\alpha snd\ f == \{\ (s,(f,s))\ |\ s.\ True\ \}$
lemma *αsnd-comp-simp*: $\alpha snd\ f\ O\ R = \{(s,(f,s'))|\ s\ s'.\ (s,s')\in R\}$ ⟨*proof*⟩

lemma *αsndI*[*simp*]: $(s,(f,s))\in\alpha snd\ f$ ⟨*proof*⟩
lemma *αsndE*: $(s,(f,s'))\in\alpha snd\ f' \Longrightarrow f=f' \wedge s=s'$ ⟨*proof*⟩

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

lemma (**in** *MFSM*) *pss-cons1*: ⟦*pss-next M A D = Some t*; *D*⊆*ps-upper M A*⟧ ⟹ (*D*,*insert t D*)∈*ps-R M A* ⟨*proof*⟩
lemma (**in** *MFSM*) *pss-cons2*: *pss-next M A D = None* ⟹ *D*∉*Domain* (*ps-R M A*) ⟨*proof*⟩

lemma (**in** *MFSM*) *pss-cons1-rev*: ⟦*D*⊆*ps-upper M A*; *D*∉*Domain* (*ps-R M A*)⟧ ⟹ *pss-next M A D = None* ⟨*proof*⟩
lemma (**in** *MFSM*) *pss-cons2-rev*: ⟦*D*∈*Domain* (*ps-R M A*)⟧ ⟹ ∃ *t*. *pss-next M A D = Some t* ∧ (*D*,*insert t D*)∈*ps-R M A*
⟨*proof*⟩

The refinement result

theorem (**in** *MFSM*) *pss-refines*: *pss-R* $\leq_{\alpha snd\ (M,A)}$ (*ps-R M A*) ⟨*proof*⟩

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*⟩

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*) ⇒ (('c,'l,'s,'m1,'m2) *pss-state*)

recdef *pss-algo-rec pss-R*$^{-1}$
pss-algo-rec ((*M*,*A*),*D*) = (*if* (*MFSM M A* ∧ *D*⊆*ps-upper M A*) *then* (*case* (*pss-next M A D*) *of None* ⇒ ((*M*,*A*),*D*) | (*Some t*) ⇒ *pss-algo-rec* ((*M*,*A*),*insert t D*)) *else* ((*M*,*A*),*D*))

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

⟦*MFSM M A*; *D*⊆*ps-upper M A*; *pss-next M A D* = *None*⟧ ⟹ *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*)
⟨*proof*⟩

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*
⟨*proof*⟩

theorem (**in** *MFSM*) *pss-correct*: *lang* (*A*⦇ δ:=*snd* (*pss-algo-rec* ((*M*,*A*),(δ *A*))) ⦈) = *pre-star* (*rules M*) *A*
⟨*proof*⟩

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* ∈ *set l* with *f e* ≠ *None*. The functional code snippet *first-that f l* corresponds to the imperative code snippet: *for e in l do* { *if f e* ≠ *None then return Some (f e)* }*; return None*

consts
first-that :: (′*s* ⇒ ′*a option*) ⇒ ′*s list* ⇒ ′*a option*
primrec
first-that f [] = *None*
first-that f (*e*#*w*) = (*case f e of None* ⇒ *first-that f w* | *Some a* ⇒ *Some a*)

lemma *first-thatE1*: *first-that f l* = *Some a* ⟹ ∃ *e*∈*set l*. *f e* = *Some a*
⟨*proof*⟩

lemma *first-thatE2*: *first-that f l* = *None* ⟹ ∀ *e*∈*set l*. *f e* = *None*
⟨*proof*⟩

lemmas *first-thatE* = *first-thatE1 first-thatE2*

lemma *first-thatI1*: $e \in set\ l \wedge f\ e = Some\ a \Longrightarrow \exists\, a'.\ first\text{-}that\ f\ l = Some\ a'$
⟨*proof*⟩

lemma *first-thatI2*: $\forall\, e \in set\ l.\ f\ e = None \Longrightarrow first\text{-}that\ f\ l = None$
⟨*proof*⟩

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

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

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

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

lemma *lookupI1*: $[\![P\ q';\ (q,w,q') \in trcl\ (set\ d)]\!] \Longrightarrow \exists\, q'.\ lookup\ P\ d\ q\ w = Some\ q'$

⟨*proof*⟩

lemma *lookupI2*: $\neg(\exists\, q'.\ P\ q' \wedge (q,w,q') \in trcl\ (set\ d)) \Longrightarrow lookup\ P\ d\ q\ w = None$
⟨*proof*⟩

lemmas *lookupE* = *lookupE1 lookupE2*
lemmas *lookupI* = *lookupI1 lookupI2*

lemma *lookup-trclAD-E1*:
assumes *map*: *set d* = *D* **and** *start*: *q*∈*Q A* **and** *cons*: *D* ⊆ *Q A* × Σ *A* × *Q A*
assumes *A*: *lookup P d q w* = *Some q′*
shows *P q′* ∧ (*q*,*w*,*q′*)∈*trclAD A D*
⟨*proof*⟩

lemma *lookup-trclAD-E2*:
assumes *map*: *set d* = *D*
assumes *A*: *lookup P d q w* = *None*
shows ¬ (∃ *q′*. *P q′* ∧ (*q*,*w*,*q′*)∈*trclAD A D*)
⟨*proof*⟩

lemma *lookup-trclAD-I1*: ⟦*set d* = *D*; (*q*,*w*,*q′*)∈*trclAD A D*; *P q′*⟧ ⟹ ∃ *q′*. *lookup P d q w* = *Some q′*
⟨*proof*⟩

lemma *lookup-trclAD-I2*: ⟦*set d* = *D*; *q*∈*Q A*; *D* ⊆ *Q A* × Σ *A* × *Q A*; ¬(∃ *q′*. *P q′* ∧ (*q*,*w*,*q′*)∈*trclAD A D*)⟧ ⟹ *lookup P d q w* = *None*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *statesE*: *q*∈*states d* ⟹ ∃ *a q′*. ((*q*,*a*,*q′*)∈*set d* ∨ (*q′*,*a*,*q*)∈*set d*) ⟨*proof*⟩
lemma *alphaE*: *a*∈*alpha d* ⟹ ∃ *q q′*. (*q*,*a*,*q′*)∈*set d* ⟨*proof*⟩

lemma *states-finite*: *finite* (*states d*) ⟨*proof*⟩
lemma *alpha-finite*: *finite* (*alpha d*) ⟨*proof*⟩

lemma *statesAlpha-subset*: *set d* ⊆ *states d* × *alpha d* × *states d* ⟨*proof*⟩

lemma *states-mono*: *set d* ⊆ *set d′* ⟹ *states d* ⊆ *states d′* ⟨*proof*⟩
lemma *alpha-mono*: *set d* ⊆ *set d′* ⟹ *alpha d* ⊆ *alpha d′* ⟨*proof*⟩

lemma *statesAlpha-insert*: *set d′* = *insert* (*q*,*a*,*q′*) (*set d*) ⟹ *states d′* = *states*

$d \cup \{q,q'\} \wedge$ *alpha* $d' =$ *insert* a (*alpha* d)
⟨*proof*⟩

lemma *statesAlpha-inv*: ⟦$q \in$*states* d; $a \in$*alpha* d; $q' \in$*states* d; *set* d'=*insert* (q,a,q') (*set* d)⟧ ⟹ *states* d = *states* d' ∧ *alpha* d = *alpha* d'
⟨*proof*⟩

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 == { $((p,\gamma,p',c'),(p\#[\gamma],a,p'\#c'))$ | $p\ \gamma\ p'\ c'\ a$. *True* }
rules-repr == { (l,l') . *rule-repr* ‘‘ *set* $l = l'$ }

lemma *rules-repr-cons*: ⟦ $(R,S)\in$*rules-repr* ⟧ ⟹ $((p,\gamma,p',c')\in$*set* $R) = (\exists\, a.\ (p\#[\gamma] \hookrightarrow_a p'\#c') \in S)$
⟨*proof*⟩

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

constdefs
find-sp $d\ s\ p$ == *first-that* ($\lambda t.$ *let* $(sh,ph,qh)=t$ *in if* $s=sh \wedge p=ph$ *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*
⟨*proof*⟩

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*⟩

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 ⇒ ('s,'c) delta ⇒ ('s × 'c × 's) option
sel-next R D ==
  first-that (λr. let (p,γ,p',c') = r in
    first-that (λt. let (q,pp',sp') = t in
      if pp'=p' then
        case find-sp D q p of
          Some spt ⇒ (case lookup (λq'. (spt,γ,q') ∉ set D) D sp' c' of
            Some q' ⇒ Some (spt,γ,q') |
            None ⇒ None
          ) | - ⇒ 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* × ('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 ⇒ None | Some t ⇒ 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}*
⟨*proof*⟩

9.3 Termination

9.3.1 Saturation upper bound

Before we can define the algorithm as recusrive function, we have to prove termination, that is well-foundedness of the corresponding recursion relation *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
seln-triv-upper R D == states D × ((fst∘snd) ‘ (set R) ∪ alpha D) × states D

lemma *seln-triv-upper-finite*: *finite (seln-triv-upper R D)* ⟨*proof*⟩

lemma *D-below-triv-upper*: *set D ⊆ seln-triv-upper R D* ⟨*proof*⟩

lemma *seln-triv-upper-subset-preserve*: *set D ⊆ seln-triv-upper A D′ ⟹ seln-triv-upper A D ⊆ seln-triv-upper A D′*
⟨*proof*⟩

lemma *seln-triv-upper-mono*: *set D ⊆ set D′ ⟹ seln-triv-upper R D ⊆ seln-triv-upper R D′*
⟨*proof*⟩

lemma *seln-triv-upper-mono-list*: *seln-triv-upper R D ⊆ seln-triv-upper R (t#D)*
⟨*proof*⟩
lemma *seln-triv-upper-mono-list′*: *x∈seln-triv-upper R D ⟹ x∈seln-triv-upper R (t#D)* ⟨*proof*⟩

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

lemma *seln-triv-upper-inv*: ⟦*t∈seln-triv-upper R D*; *set D′ = insert t (set D)*⟧ ⟹ *seln-triv-upper R D = seln-triv-upper R D′*
⟨*proof*⟩

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

lemma *find-sp-in-states*: *find-sp D s p = Some qh* $\Longrightarrow$ *qh*∈*states D* ⟨*proof*⟩

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*∉*set D* ∧ *t*∈*seln-triv-upper R D*
⟨*proof*⟩

Hence, it does not change the upper bound

corollary *sel-next-upper-preserve*: ⟦*sel-next R D = Some t*⟧ $\Longrightarrow$ *seln-triv-upper R D = seln-triv-upper R (t#D)* ⟨*proof*⟩

9.3.2 Well-foundedness of recursion relation

lemma *seln-R-wf*[*recdef-wf*]: *wf* (*seln-R*$^{-1}$) ⟨*proof*⟩

9.3.3 Definition of recursive function

consts
pss-algo-rec :: (′*c*,′*s*) *seln-state* ⇒ (′*c*,′*s*) *seln-state*

recdef *pss-algo-rec seln-R*$^{-1}$
pss-algo-rec (*R*,*D*) = (*case sel-next R D of Some t* ⇒ *pss-algo-rec* (*R*,*t#D*) | *None* ⇒ (*R*,*D*))

lemma *pss-algo-rec-newsimps*[*simp*]:
⟦*sel-next R D = None*⟧ $\Longrightarrow$ *pss-algo-rec* (*R*,*D*) = (*R*,*D*)
⟦*sel-next R D = Some t*⟧ $\Longrightarrow$ *pss-algo-rec* (*R*,*D*) = *pss-algo-rec* (*R*,*t#D*)
⟨*proof*⟩

declare *pss-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-ex*) *sel-nextE1*:
assumes *A*: *sel-next R D = Some* (*s*,*γ*,*q*′)
shows (*s*,*γ*,*q*′)∉*set D* ∧ (∃ *q p a c*′. *s*=*sp A q p* ∧ [*p*,*γ*]$\hookrightarrow_a$ *c*′ ∈ *rules M* ∧ (*q*,*c*′,*q*′)∈*trclAD A* (*set D*))
⟨*proof*⟩

lemma (**in** *MFSM-ex*) *sel-nextE2*:
  assumes *A*: *sel-next R D = None*
  shows $\neg(\exists\ q\ p\ \gamma\ q'\ a\ c'\ t.\ t \notin set\ D \land t=(sp\ A\ q\ p,\gamma,q') \land [p,\gamma]\hookrightarrow_a c' \in rules\ M \land (q,c',q') \in trclAD\ A\ (set\ D))$
⟨*proof*⟩

lemmas (**in** *MFSM-ex*) *sel-nextE = sel-nextE1 sel-nextE2*

lemma (**in** *MFSM-ex*) *seln-cons1*: $[\![sel\text{-}next\ R\ D = Some\ t]\!] \Longrightarrow (set\ D, insert\ t\ (set\ D)) \in ps\text{-}R\ M\ A$ ⟨*proof*⟩
lemma (**in** *MFSM-ex*) *seln-cons2*: $sel\text{-}next\ R\ D = None \Longrightarrow set\ D \notin Domain\ (ps\text{-}R\ M\ A)$ ⟨*proof*⟩

lemma (**in** *MFSM-ex*) *seln-cons1-rev*: $[\![set\ D \notin Domain\ (ps\text{-}R\ M\ A)]\!] \Longrightarrow sel\text{-}next\ R\ D = None$ ⟨*proof*⟩
lemma (**in** *MFSM-ex*) *seln-cons2-rev*: $[\![set\ D \in Domain\ (ps\text{-}R\ M\ A)]\!] \Longrightarrow \exists\ t.\ sel\text{-}next\ R\ D = Some\ t \land (set\ D, insert\ t\ (set\ D)) \in ps\text{-}R\ M\ A$
  ⟨*proof*⟩

DPN-specific abstraction relation, to associate states of deterministic algorithm with states of *ps-R*

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

lemma *αselnI*: $[\![S=set\ D;\ MFSM\text{-}ex\ M\ A\ R\ D]\!] \Longrightarrow (S,(R,D)) \in \alpha seln\ M\ A$
  ⟨*proof*⟩

lemma *αselnD*: $(S,(R,D)) \in \alpha seln\ M\ A \Longrightarrow S=set\ D \land MFSM\text{-}ex\ M\ A\ R\ D$
  ⟨*proof*⟩

lemma *αselnD'*: $(S,C) \in \alpha seln\ M\ A \Longrightarrow S=set\ (snd\ C) \land MFSM\text{-}ex\ M\ A\ (fst\ C)\ (snd\ C)$ ⟨*proof*⟩

lemma *αseln-single-valued*: $single\text{-}valued\ ((\alpha seln\ M\ A)^{-1})$
  ⟨*proof*⟩

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

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*: $graph\ sel\text{-}next\text{-}state \leq_{Id} seln\text{-}R$ ⟨*proof*⟩

interpretation *det-impl*: *detRef-impl pss-algo-rec sel-next-state seln-R*
  ⟨*proof*⟩

And then infer correctness of the deterministic algorithm

theorem (**in** *MFSM-ex*) *pss-correct*:

assumes *D-init*: *set D* = *δ A*
shows *lang* (*A*⦇ *δ*:=*set* (*snd* (*pss-algo-rec* (*R*,*D*))) ⦈) = *pre-star* (*rules M*) *A*
⟨*proof*⟩

corollary (**in** *MFSM*) *pss-correct*:
assumes *repr*: *set D* = *δ A* (*R*,*rules M*)∈*rules-repr*
shows *lang* (*A*⦇ *δ*:=*set* (*snd* (*pss-algo-rec* (*R*,*D*))) ⦈) = *pre-star* (*rules M*) *A*
⟨*proof*⟩

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.