Certification dans Isabelle d'un Algorithme d'Accessibilité pour les Systèmes à Pile

Raphaël Cauderlier, Tobias Nipkow, Javier Esparza, TU München

2011

Le contexte général

Les systèmes de transition à pile (Pushdown systems en anglais) sont une modélisation naturelle pour la sémantique des programmes, pas nécessairement déterministes, écrits dans des langages séquentiels (C ou java par exemple) dans lesquels les fonctions peuvent s'appeller mutuellement et récursivement et les variables prennent leurs valeurs dans un domaine fini.

Beaucoup de problèmes de vérification se ramènent au problème de l'accessibilité d'un ensemble (éventuellement infini) de configurations. Une approche efficace pour résoudre ce problème dans le cas des systèmes à piles consiste à raisonner en remontant les transitions : étant donné un ensemble de configurations cible C, on cherche à calculer (efficacement) l'ensemble $\mathbf{pre}^*(C)$ des configurations qui mènent, en suivant un nombre arbitraire de transitions, à un élément de C. C'est possible dans le cas où C est un ensemble rationnel de configurations donné par un automate [].

Le problème étudié

L'algorithme de calcul de $\mathbf{pre}^*(C)$ pour un langage rationnel C est implémenté dans des model-checkers tel que [], les utilisations de ce genre d'outils étant souvent très sensibles (médecine, aéronotique, spatial...), il est impératif que l'on ait totalement confiance dans cet algorithme. Les assistants de preuve comme Isabelle apportent la plus grande confiance possible aujourd'hui. Mon travail n'est pas totalement nouveau, Peter Lammich à Münster a travaillé sur des systèmes un peu plus généraux et certifié dans Isabelle la correction de l'algorithme de calcul de $\mathbf{pre}^*$ pour ces systèmes mais n'a pas encore publié ses résultats.

La contribution proposée

Tout mon travail a été réalisé à l'intérieur du système Isabelle. J'ai écrit une formalisation des systèmes à pile et des automates reconnaissant des ensembles de configurations de système à pile, j'ai certifié l'algorithme abstrait non déterministe présenté dans [] et j'en ai déduit, toujours à l'intérieur d'Isabelle, le théorème suivant : "pour tout langage rationnel C, $\mathbf{pre}^*(C)$ est rationnel".

Les arguments en faveur de sa validité

Ma certification est entièrement vérifiable par Isabelle (version 2011), si le noyau logique d'Isabelle est correct alors ma certification l'est également. Mes résultats sont au niveau de confiance maximal possible par la science actuelle.

Par ailleurs, pour faciliter la réutilisation de mes preuves, j'ai pris soin de toutes les écrire dans le langage **Isar** d'Isabelle qui les rend beaucoup plus lisibles et faciles à maintenir que des preuves écrites en "apply-style".

Enfin mes résultats sont très abstraits et généraux, je ne force presque aucun choix d'implémentation.

Le bilan et les perspectives

Il y a encore un trou important entre ma formalisation très abstraite et les algorithmes réellement utilisés dans les outils de Model-chacking comme [], c'est le domaine de raffinement de donnée qui est un sujet qui intéresse beaucoup Peter Lammich et l'équipe dans laquelle j'ai fait mon stage.

Il est aussi possible de symétriser le résultat en cherchant les configurations accessibles depuis un ensemble donné de configurations, c'est un peu plus technique car il faut ajouter des états aux automates et les marquer mais l'essentiel du travail est une adaptation de mes résultats.

Certifying in Isabelle an Automata-Based Reachability Algorithm for Pushdown Systems

Raphaël Cauderlier

2011

Abstract

Pushdown systems are used in Model-Checking to represent abstracted, non-determinist programs. For this class of infinite transition systems, reachability of regular sets of configurations is computable by automata transformation. I proved this in the proof assistant Isabelle.

Contents

1 Introduction

When proving properties of a program, one way to drastically reduce the domain of the variables (and so the size of the search space) is to abstract from the variable values to get an approximation of the behaviour of the original for which Model-Checking is easy to perform. For example, to prove mutual exclusion for a system using locks, it is usually OK to ignore the value of every thing but locks because for this precise problem, we are not interested in what computation is actually done in the critical section. Abstraction often leads to non-determinism because we abstract over the value of variables used in tests. For this we need good tools to handle non-deterministic transition systems.

Pushdown systems used in the Moped [2] model-checker are very good at modeling sequential programs once abstracted enough to get a reasonable domain size for variables. Despite their infinite space, reachability properties can be computed efficiently using finite automata [1, 6]. Computing reachability for some classes of infinite sets of configurations is even possible.

Because of their use in Model-Checking, we want to trust these reachability algorithms by certifying them in the most formal way possible: using a proof assistant.

Isabelle [5] is one of them, its main advantage is the readability of its proofs written in the declarative Isar language [7]. My internship happened in Technische Universität München in the German part of Isabelle development team, the Theorem Proving Group lead by Tobias Nipkow in collaboration with Javier Esparza, also professor at TUM, for the Model-Checking part.

2 Scientific Context of my Internship

2.1 Isabelle

2.1.1 Proof Assistant

Isabelle is a proof assistant. This means that Isabelle looks like a programming environment for a functional language like OCaml [3] but instead of producing programs, it produces proofs. Every step of every proof in Isabelle is verified according to a small logic kernel which we trust so all theorems proved in Isabelle are logical consequences of the axioms introduced in Isabelle libraries.

Working with a proof assistant is a way to increase the confidence we can have in a proof. Formalizing the results of a scientific article often reveals

forgotten cases or even logical flaws (see for example [4] page 147).

Isabelle knows several logics but I only used the default one : Higher Order Logic [7]. HOL is a typed logic in which one can quantify over arbitrary (well typed) lambda terms.

In HOL one can define datatypes and recursive function pretty much like in a pure functional programming language but all functions must terminate. Most of the time, the termination proof is an obvious induction following a datatype induction principle but the user can also provide it's own complex termination proof if it is not found automatically by the system.

2.1.2 Isar Language

Any proof in Isabelle can be written either in so called apply-style (as a list of **apply** commands which reduce the current goal until it becomes *True*) or using the Isar language. Apply-style proofs are easy to write but very hard to read and maintain. Isar proofs on the other hand use usual English logic words (**assume**, **show**, **moreover**, **hence** ...) which make them easier to read for humans. Also increasing the readability is the use of the declarative style. This means that proofs in Isar progress from assumptions to goals and that the user writes each property he proves before actually providing a method or a subproof for it.

Let's look at an example from group theory : the characterization of the subgroup generated by a set of elements.
This theorem states that both definitions of the subgroup generated by a set S are equivalent.

$$grp\ S = grp'\ S$$

where *grp* is defined by $grp\ S = \bigcap \{A \mid subgroup\ A \wedge S \subseteq A\}$ and grp' is defined by $grp'\ S = \{\Pi\ as \mid pow\text{-}prod\ as\ S\}$
whith $pow\text{-}prod\ as\ S \equiv \forall\, a{\in}as.\ \exists\, b{\in}S.\ \exists\, z.\ a = b^z$

theorem *isar-example*: $grp\ S = grp'\ S$
proof
— the keyword **proof** starts a proof by applying a standard method
— here it transforms the set equality into two inclusions
have $S \subseteq grp'\ S$
— to prove the first one $grp\ S \subseteq grp'\ S$, we first prove this
proof
— $S \subseteq grp'\ S$ has been turned to $\bigwedge a.\ a \in S \Longrightarrow a \in grp'\ S$
fix a
assume $a \in S$
thus $a \in grp'\ S$
using *grp'-def*[*of* [*a*], *OF pow-prod-singl*] **by** *simp*

```
  qed
  with subgroup-grp'[of S] show grp S ⊆ grp' S
    unfolding grp-def by blast
next
  show grp' S ⊆ grp S
  proof
    fix a
    assume a ∈ grp' S
    then obtain as where p: pow-prod as S and a: a = Π as
      unfolding original-grp'-def by blast
    have set as ⊆ grp S
    proof
      fix x
      assume x ∈ set as
      with p obtain b z where b: b ∈ S and [simp]: x = b^z
        — for later use, propositions can be explicitly named
        — or declared as simplification rules
        unfolding pow-prod-def by blast
      thus x ∈ grp S using pow-grp[OF b] by fast
    qed
    from subgroup-prod[of grp S as, OF - this] show a ∈ grp S
      by (simp add: a)
  qed
qed
```

2.2 Definitions

2.2.1 Pushdown Systems

Pushdown systems [1] are like pushdown automata ignoring their input. They are transition systems which use an unbounded stack so their set of configurations are usually infinite. Pushdown systems are good for modeling sequential programs written in language such as C or java where functions or procedures can call each other, sometimes recursively or mutually recursively because this is really how these programs are compiled. However, this model forces finiteness of the variable domain and the size of the pushdown system will be proportional to it.

Definition 1 (Pushdown System) *A pushdown system $\mathcal{P}$ is formally defined by:*

- *A finite set Control $\mathcal{P}$ of control states.*
- *An alphabet Σ $\mathcal{P}$ of stack symbols.*
- *An initial configuration Initial $\mathcal{P}$ (which doesn't play any role for our purpose)*

- *A finite set $\Delta\ \mathcal{P}$ of transition rules of the following form:*

$$(p,\ \gamma) \hookrightarrow (p',\ w)$$

where p and p' are control states, γ is a stack symbol and w is a string of stack symbols.

We fix a pushdown system $\mathcal{P}$.

Definition 2 (Configuration) *Configurations of $\mathcal{P}$ are couples of control states and stack contents (i.e. strings of stack symbols).*

Definition 3 (Transition) *Transitions of $\mathcal{P}$ are defined by:*
$(p, \gamma u) \Rightarrow (p', w\ @\ u)$ if and only if $(p, \gamma) \hookrightarrow (p', w)$ is a rule of the system where u is an arbitrary string of stack symbols.

As usual, we denote by $\Rightarrow^*$ the reflexive transitive closure of the transition relation $\Rightarrow$.

2.2.2 Reachability, pre^*, $post^*$

In order to compute reachability properties for pushdown systems, two functions, pre^* and $post^*$ have proved to be very practical [1, 6]. They are defined by

$$pre^*\ C = \{c \mid \exists\, c' \in C.\ c \Rightarrow^*\ c'\}$$

and symmetrically

$$post^*\ C = \{c \mid \exists\, c' \in C.\ c' \Rightarrow^*\ c\}$$

The reachability problem

$$\begin{array}{c}\text{given two configurations } c_1 \text{ and } c_2,\\ \text{do we have } c_1 \Rightarrow^*\ c_2?\end{array}$$

can be expressed in term of pre^* (resp. $post^*$) as

$$\begin{array}{c}\text{given two configurations } c_1 \text{ and } c_2,\\ \text{do we have } c_1 \in pre^*\ \{c_2\} \text{ (resp. } c_2 \in post^*\ \{c_1\})?\end{array}$$

So we are looking for ways to represent (possibly infinite) sets with the following requirements:

- one-element sets must be representable
- membership test must be computable (and preferably easy to perform)
- if a set C is representable, then $pre^*\ C$ is also representable and we have an algorithm transforming any representation of C into a representation of $pre^*\ C$

They are several way to solve this [1], I only worked on the most simple one: regular sets and automata.

2.2.3 $\mathcal{P}$-automata

So we are going to manipulate sets of configuration represented by automata. Since the configurations are not just strings but couples, we will slightly change the usual definition of automata and recognition by taking into account the initial state used to read the word.

Definition 4 ($\mathcal{P}$-automaton) *A $\mathcal{P}$-automaton $\mathcal{A}$ is defined by:*

- *a finite set $Q\ \mathcal{A}$ of states containing the set Control $\mathcal{P}$ of control states of $\mathcal{P}$ used as initial states*
- *a finite set Final $\mathcal{A}$ of final states*
- *a finite set $\delta\ \mathcal{A}$ of transitions i.e. triples of state, stack symbol and state. $(p,\ \gamma,\ q) \in \delta\ \mathcal{A}$ is abbreviated by $p \xrightarrow{\gamma}_{\mathcal{A}} q$.*

Since $\mathcal{P}$ is fixed, we will simply call them automata. The transition relation $\delta\ \mathcal{A}$ is extended the usual way to strings.

Definition 5 (Language)
The language of $\mathcal{A}$ is then defined by replacing the usual set of initial states by Control $\mathcal{P}$ and taking the chosen control state into account.

$$\mathcal{L}\ \mathcal{A} \equiv \{(p,\ w) \mid \exists\, q{\in}Final\ \mathcal{A}.\ p \in Control\ \mathcal{P} \wedge p \xrightarrow{w}_{\mathcal{A}} q\}$$

2.3 Main Algorithm and Theorem

2.3.1 pre^* of a Regular Language

The main property of pre^* and $post^*$ for pushdown systems is that they preserve the regularity ; for any regular set C of configurations (of $\mathcal{P}$), $pre^*\ C$ and $post^*\ C$ are also regular [1].

2.3.2 Saturation Procedure Computing pre^*

Even more, we actually have a procedure to transform an automaton recognizing C into an automaton recognizing $pre^*\ C$ (and also for $post^*\ C$ but this is a bit more complex). This work by saturating the following rule:

If $(p,\ \gamma) \hookrightarrow (p',\ w)$ is a rule of $\mathcal{P}$ and $p' \xrightarrow{w}_{\mathcal{A}} q$, then add a transition $p \xrightarrow{\gamma}_{\mathcal{A}} q$.

3 My Formalization

3.1 Typing

Strings are simply represented by lists. This is a very reasonable typing choice for a lot of reasons:

- This is a very simple datatype for which inductive proofs are usually easy.
- It's also very easy to define recursive functions on lists and reason about them inductively.
- Isabelle provides a lot of theorems for lists in the standard library
- When they are used as stack values, it's reasonable to access only the head in constant time.

I tried not to force any other implementation choice so the choice of types is totally let to the user. I accomplished this by working with polymorphic datatypes using a type variable $'s$ for stack symbols and automata letters and another type variable $'c$ for pushdown systems control symbols and automata states. So for example, given an automaton $\mathcal{A}$, it's transition table is so typed :

$\delta\ (\mathcal{A} :: ('c,\ 's)\ P\text{-}Automaton) :: ('c * 's * 'c)\ set$

However, for technical reasons that we will see later, I had to restrict the choice for $'c$ to an infinite type and my results are really only totally proved when $'c$ is the type of natural numbers.

3.1.1 Finite Sets

I formalized pushdown systems and automata using the built-in *set* type constructor and constraining them to be finite using the *finite* predicate. This level of abstraction permits to write implementation-independent proofs which are often easier to write and more general. The price to pay for it is the loss of computability at this level of abstraction. Automatically deriving for such theorems corresponding ones replacing all the finite sets by a real (usable for computation) datatype is the goal of a branch of theorem proving called Data Refinement and on which several Isabelle developers and contributors are currently working.

3.1.2 Records

Pushdown systems and automata are defined using the Isabelle record mechanism :

type-synonym $('c, 's)$ *config* $= 'c * 's$ *list*

record $('c, 's)$ *Pushdown-System* $=$
  $\Delta :: ('c * 's * 'c * 's$ *list*$)$ *set*
  Initial $:: ('c, 's)$ *config*
  $\Sigma :: 's$ *set*
  Control $:: 'c$ *set*

record $('c, 's)$ *P-Automaton* $=$
  $Q :: 'c$ *set*
  $\delta :: ('c * 's * 'c)$ *set*
  Final $:: 'c$ *set*

I have chosen this mechanism because it gives explicit names to fields which makes proofs easier to read and maintain than defining them by tuples (which in Isabelle are just nested couples).

I did however not use the full power of records because I did not want nor need to cope with exstensibility; usually, fields can be added to records in a way similar to classe inheritance in object-oriented programming.

3.2 Locales

Locales are the way to define persistent contexts in Isabelle. I found them very useful.

I use locales to axiomatize structural constraints which defines my objects, especially finiteness constraints. For instance, here is the definition of the locale predicate for automata:

$PA\ \mathcal{P}\ \mathcal{A} \equiv$

$(PS\ \mathcal{P} \wedge finite\ (Q\ \mathcal{A})) \wedge$
$\Delta\ \mathcal{A} \subseteq Q\ \mathcal{A} \times \Sigma\ \mathcal{P} \times Q\ \mathcal{A} \wedge Final\ \mathcal{A} \subseteq Q\ \mathcal{A} \wedge Control\ \mathcal{P} \subseteq Q\ \mathcal{A}$

For pushdown systems, I use the locale PS in order to fix a working pushdown system $\mathcal{P}$, this is easy because I have only one pushdown system to manage.

For automata on the other hand it would be harder to fix one because my important lemmas deal with several automata so I'm working outside from the locale PA but it nevertheless gives me a useful predicate PA which summarizes all finiteness constraints over the fields of my automata. For the small lemmas dealing with just one automaton, it would probably be cleaner to write everything in the locale itself.

The locale mechanism also surprisingly solved a typing problem: usually in abstract automata formalisations, it's quite hard to define what a regular set is. If we take the following definition

definition *regular* :: *'letter set* ⇒ *bool*
  where *regular C* ≡ (∃ (*A* :: (*'letter*, *'state*) *automaton*). *C* = *L A*)

then Isabelle complains because the type of the states of $\mathcal{A}$ is unrelated to C so it cannot be infered from the predicate argument.

Of course the type of the states of $\mathcal{A}$ in this definition does not really matter unless it bounds the number of states, so a possible workaround is to impose a type in this definition of regular like this:

definition *regular'* :: *'letter set* ⇒ *bool*
  where *regular' C* ≡ (∃ (*A* :: (*'letter*, *nat*) *automaton*). *C* = *L A*)

The difference between the two definitions is the use of *nat* instead of the type variable *'state*. *nat* is not a type variable, it is the type of natural numbers defined in HOL standard library. This workaround however requires more work if we want to avoid forcing any implementation choice so we should prove a theorem like

theorem *arbitrary-state-type*:
    ∀ (*A* :: (*'letter*, *'state*) *automaton*).
      ∃ (*A'* :: (*'letter*, *nat*) *automaton*).
        L A = *L A'*

from which we finally can deduce simple facts like

corollary *regular-language*: ∀ (*A* :: (*'letter*, *'state*) *automaton*). *regular'* (*L A*)
  by (*simp add*: *regular'-def arbitrary-state-type*)

This is not very modular, it only works because there is a universal (independent of C) choice for the type variable *'state* moreover an additionnal proof has to be supplied.

My definition of regularity is simply

definition *myregular* :: ('*c*, '*s*) *config set* ⇒ *bool*
  where *myregular C* ≡ (∃ *A*. *PA P A* ∧ *C* = *L A*)

It would work even if $\mathcal{L}$ just returned a *'s set* as it usually does because all type variables of $\mathcal{A}$ are fixed by those of $\mathcal{P}$, which is a constant in the locale context in which regularity is defined. In other words, I just define $\mathcal{P}$-regularity for which the typing issue of regularity does simply not exist.

Of course this magic is also limitating; the modelization of $\mathcal{P}$-automata in [1] does not force the set of states of $\mathcal{P}$-automata to be of the same type than *Control P* but to provide for each $\mathcal{P}$-automaton $\mathcal{A}$ a bijection between the set of initial states of $\mathcal{A}$ and *Control P*. This is not possible in my modelization.

3.3 Typeclass for Fresh Copy

I needed to be able to add fresh states to my automata so I formalized the notion of a type *'a* together with a copy function which take two arguments of type *'a set*, assumed to be finite, the source to copy *source* and a set of forbidden values *forbidden* and returns a function f of type *'a* ⇒ *'a* such that f is injective on *source* and the image of *source* under f is disjoint from *source* ∪ *forbidden*.

Isabelle provides a mecanism to define types together with constants and specifications, it is borrowed from Haskell and called a typeclass.

class *copy* =
  fixes *copy-function* :: *'a set* ⇒ *'a set* ⇒ (*'a* ⇒ *'a*)
  assumes *finite-copy*:
    ⟦*finite forbidden*; *finite source*⟧ ⟹
      ((*copy-function forbidden source*) ‘ *source* ∩ (*source* ∪ *forbidden*) = ∅) ∧
      inj-on (*copy-function forbidden source*) *source*

A typeclasses is simultaneously a kind of type for types together with a locale; it is possible to work in a context named *copy* exactly as if *copy* had been defined as a locale and it is enough to mark a type variable by the typeclass like this: *'a* :: *copy* to assume that *'a* is an instance of the typeclass and get access to the defined constants, assumptions and lemmas proved in the context. so in fact the real definition of the record for automata is

record ('*c* :: *copy*, '*s*) *P-automaton* =
  Q :: *'c set*
  δ :: (*'c* ∗ *'s* ∗ *'c*) *set*
  Final :: *'c set*

For practical use, typeclasses are instanciated. When defining a typeclass in Isabelle, it is good to provide one explicit instatiation or to prove it to be a superclass of some nonempty typeclass.

The typeclass *copy* can be instanciated by any infinite type but I only proved *nat* to be an instance of it. This does not force the user to implement automata which use *nat* for state type, he just has to provide a proof that his state type is an instance of the typeclass *copy* i.e. to give an explicit *copy-function* for his type and to prove it to be conform to the corresponding *finite-copy* specification.

4 My Certification

My certification of the algorithm for computing pre^* for regular sets of configurations is essentially an adaptation of Stefan Swoon's proof [6].

4.1 Non-determinist Abstract Algorithm

There are to main limitations to the computability of my algorithm which explain why I can not run it on an example (which would nicely decorate this report but also be very interesting for a practical point of view for applications of my certification).

The first reason is that I use Isabelle built-in *set* types to represent sets and taking an element of a set is not a deterministic action.

The second reason is that my algorithm (the saturation procedure) is in essence not deterministic. Let's look at the rule again:

> If $(p,\ \gamma) \hookrightarrow (p',\ w)$ is a rule of $\mathcal{P}$ and $p' \xrightarrow{w}_{\mathcal{A}} q$, then add a transition $p \xrightarrow{\gamma}_{\mathcal{A}} q$.

It does not tell us in what order new transitions should be added and in fact efficiency of the different implementations is very dependent on the way matching $(p,\ \gamma,\ p',\ w,\ q)$ tuples are found.

The way I defined it should be compatible with all possible implementations; I defined it as a transition $\mathcal{A} \mapsto \mathcal{A}'$ if and only if $\mathcal{A}'$ can be obtain from $\mathcal{A}$ by applying the saturation rule exactly once:

$$\mathcal{A} \mapsto \mathcal{A}\ (\!|\delta := \mathit{insert}\ (p,\ \gamma,\ q)\ (\delta\ \mathcal{A})|\!) \equiv$$
$$\exists\, p\ p'\ q\ \gamma\ w.\ \mathit{PA}\ \mathcal{P}\ \mathcal{A} \wedge p \in \mathit{Control}\ \mathcal{P} \wedge \gamma \in \Sigma\ \mathcal{P} \wedge q \in Q\ \mathcal{A} \wedge p' \in$$
$$\mathit{Control}\ \mathcal{P} \wedge (p,\ \gamma) \hookrightarrow (p',\ w) \wedge p' \xrightarrow{w}_{\mathcal{A}} q \wedge (p,\ \gamma,\ q) \notin \delta\ \mathcal{A}$$

4.2 Contrast Between Proofs for Humans and Proofs for Computers

4.2.1 Inductive Principles

In Isabelle, every inductive definition automatically produces a set of inductive principles which are in practise more convenient to use than comming

back to induction on the size (or length) of the object we are studying. This contrasts with pen-and-paper proofs for which inductions are often done on natural numbers. For example, the following piece of proof from [6]:

> **Lemma 3.1** *For every configuration* $(p,\ v) \in \mathcal{L}\ \mathcal{A}$, *if* $(p',\ w) \Rightarrow^*\ (p,\ v)$ *then* $p' \xrightarrow{w}_{\mathcal{A}'} q$ *for some final state* q *of* $\mathcal{A}'$.
> **Proof:** Assume $(p',\ w) \Rightarrow^k\ (p,\ v)$. We proceed by induction on k.

becomes

```
lemma Lemma3-1:
  assumes PA 𝒫 𝒜
  assumes 𝒜 ↦* 𝒜'
  assumes sat 𝒜'
  assumes (p', w) ⇒* (p, v)
  assumes (p, v) ∈ ℒ 𝒜
  shows ∃ q ∈ PA-rec.Final 𝒜'. 𝒜' ⊢ p' −w→* q
  using assms(4)
proof (induct rule: converse-rtrancl-induct2)
```

in my Isabelle formalisation. This means that I am doing a proof using the following induction principle (defined in Isabelle library):

$$\frac{a \Rightarrow^*\ b \qquad P\ b \qquad \bigwedge y\ z.\ \dfrac{y \Rightarrow\ z \qquad z \Rightarrow^*\ b \qquad P\ z}{P\ y}}{P\ a}$$

from assumption number 4 : $(p',\ w) \Rightarrow^*\ (p,\ v)$.

4.2.2 "Without loss of generality, we assume that $\mathcal{A}$ has no transition leading to an initial state"

This sentence (from [6], page 32), is perfectly fine in a scientific article, this assumption sounds very reasonable and I think it is despite I spent one month proving it.

"Without loss of generality"-properties are evil for theorem proving. For this one, I had to write an algorithm for converting any automaton $\mathcal{A}$ to an automaton $\mathcal{A}'$ recognizing the same language and having no transition leading to an initial state. Once again, this is not very hard to find and prove with pen and paper but it takes one third of my code file (see the appendix). The idea is to get a copy of all initial states, use the read letter as a bridge to this parallel universe and stay in it until the end of input.

Formally, I do this by this function:

$$
\begin{array}{l}
\textit{double-init}\ \mathcal{A} = \\
(\!|\, Q = Q\ \mathcal{A} \cup \textit{fcopy}\ \mathcal{A}\ \text{`}\ \textit{Control}\ \mathcal{P}, \\
\quad \Delta = \{(p, \gamma, \textit{gcopy}\ \mathcal{A}\ q) \mid (p, \gamma, q) \in \delta\ \mathcal{A} \wedge p \in \textit{Control}\ \mathcal{P}\} \cup \\
\qquad \{(\textit{gcopy}\ \mathcal{A}\ q, \gamma, \textit{gcopy}\ \mathcal{A}\ q') \mid (q, \gamma, q') \in \delta\ \mathcal{A}\}, \\
\quad \textit{Final} = \textit{Final}\ \mathcal{A} \cup \textit{gcopy}\ \mathcal{A}\ \text{`}\ \textit{Final}\ \mathcal{A} |\!)
\end{array}
$$

The first line extends the set of states by a fresh copy of initial states (this is the reason why I needed to formalize the notion of fresh copy), *fcopy* is the fresh function injective on $\textit{Control}\ \mathcal{P}$ and avoiding $Q\ \mathcal{A}$: $\textit{fcopy}\ \mathcal{A} = \textit{fresh-copy}\ (Q\ \mathcal{A})\ (\textit{Control}\ \mathcal{P})$. The function *gcopy* is *fcopy* extended to all $Q\ \mathcal{A}$ by identity on non initial states, this is the translation the parallel universe.

The second line is the "bridge" part and the third line the "parallel" part. The last line is not simply $\textit{Final} = \textit{gcopy}\ \mathcal{A}\ \text{`}\ \textit{Final}\ \mathcal{A}$ as one could think because we have to deal with the empty word which is too short to access the parallel universe.

And then the real proofs begin... By the way, avoiding to have to add knew states (even one by one which gives simpler proofs) is exactly what makes the certification (termination and correctness) of the algorithm computing pre^* easier than the one computing $post^*$ so at least all this work can help adapting my work to the certification for $post^*$.

5 Conclusion

A lot of work is still to be done on this subject. To adapt it to $post^*$ should not be very difficult now I have formalised the notion of fresh copy and research on data refinement is very active so I'm sure that my code will soon be able to certify real implementations without too much efforts. It then could be generalized to a lot of other classes of sets of configurations for cases where the set we want to compute reachability for is not regular because proofs for other classes of sets are seldom very different from the one for regular sets.

6 Acknowledgement

I want to thank my supervisors Tobias Nipkow and Javier Esparza for this very interesting internship topic and my tutor in Cachan Stefan Schwoon for having asked this internship for me.

I have felt really welcome by the Theorem Proving Group, I'm really grateful for the ambiance and the precision and pertinence of the answers to my questions concerning Isabelle usage and automata formalization.

Thanks also to Peter Lammich for having sent to me his unpublished modelization of Dynamic Pushdown Networks in Isabelle which helped me a lot for a crucial lemma and discussed with me automata formalization issues.

A Source code

theory *Pushdown-systems*
imports *Main*
begin

type-synonym *'a rel* = (*'a* ∗ *'a*) *set*
— relation type synonym

abbreviation *epsilon* :: *'a list* (*ε*)
  where *ε* ≡ []
— strings are implemented by lists, the empty word is denoted by *ε*

A.1 Fresh Function and Copy

We will need to add fresh states to automata, for this purpose we define a class of types enriched with a copy functions wich returns finitely many fresh elements.

class *fresh* =
  fixes *fresh-copy* :: *'a set* ⇒ *'a set* ⇒ (*'a* ⇒ *'a*)
  assumes *finite-fresh-copy*:
    ⟦*finite forbidden-set*; *finite B*⟧ ⟹ ((*fresh-copy forbidden-set B*) ‘ *B* ∩ (*B* ∪ *forbidden-set*) = {}) ∧ *inj-on* (*fresh-copy forbidden-set B*) *B*

begin
end

Any infinite type is a member of this class but for our purpose, using *nat* will be enough.

instantiation *nat* :: *fresh*
begin
definition *supnat* :: *nat set* ⇒ *nat*
  where *supnat A* = *fold max 0 A*

lemma *sup*: *finite A* ⟹ *x* ∈ *A* ⟹ *x* ≤ *supnat A*
  unfolding *supnat-def max-def* **by** (*induct A rule*: *Finite-Set.finite-induct*) *auto*

definition *fresh-copy-def*:
 fresh-copy forbidden-set B k = *1* + *supnat* (*forbidden-set* ∪ *B*) + *k*

instance proof
  fix *forbidden-set B* :: *nat set*
  assume *finite*: *finite forbidden-set finite B*
  let *?n* = *1* + *supnat* (*forbidden-set* ∪ *B*)
  have [*simp*]: ⋀*k*. *fresh-copy forbidden-set B k* = *?n* + *k*

```
    by (auto simp: fresh-copy-def)
  show fresh-copy forbidden-set B ‘ B ∩ (B ∪ forbidden-set) = {} ∧
       inj-on (fresh-copy forbidden-set B) B
  proof (auto simp: inj-on-def)
    fix k
    assume Suc (supnat (forbidden-set ∪ B) + k) ∈ B
    hence ?n + k ∈ B by simp
    hence ?n + k < ?n using finite sup[of forbidden-set ∪ B ?n + k] by simp
    thus False by simp
  next
    fix k
    assume Suc (supnat (forbidden-set ∪ B) + k) ∈ forbidden-set
    hence ?n + k ∈ forbidden-set by simp
    hence ?n + k < ?n using finite sup[of forbidden-set ∪ B ?n + k] by simp
    thus False by simp
  qed
qed
end
```

A.2 Pushdown Systems

Pushdown systems are transition systems over a finite set of control symbols using an unbounded stack. A configuration of a pushdown systemm is given by a control symbol and a string of stack symbols.

```
type-synonym ('c, 's) config = 'c * 's list
```

Without loss of generality, we consider pushdown systems which pop one symbol and push at most two symbols; this would be useful for $post^*$.

$'c$ stands for control symbol and $'s$ for stack symbol.

```
record ('c :: fresh, 's) PS-rec =
  Δ0 :: ('c * 's * 'c) set
  Δ1 :: ('c * 's * 'c * 's) set
  Δ2 :: ('c * 's * 'c * 's * 's) set
  Initial :: ('c, 's) config
  Σ :: 's set
  Control :: 'c set
```

A.3 Transitions in a Pushdown System

Membership and finiteness constraints are expressed by the following locale in wich all interesting work is going to be done.

```
locale PS =
  fixes P :: ('c :: fresh, 's) PS-rec
  assumes finite (Σ P)
  and finite (Control P)
  and Δ0 P ⊆ Control P × Σ P × Control P
  and Δ1 P ⊆ Control P × Σ P × Control P × Σ P
```

and $\Delta_2\ \mathcal{P} \subseteq \mathit{Control}\ \mathcal{P} \times \Sigma\ \mathcal{P} \times \mathit{Control}\ \mathcal{P} \times \Sigma\ \mathcal{P} \times \Sigma\ \mathcal{P}$
begin

fun *PS-rule* :: $('c * 's) \Rightarrow ('c, 's)\ \mathit{config} \Rightarrow \mathit{bool}$
(**infix** $\hookrightarrow$ *100*)
where
$(p, \gamma) \hookrightarrow (p', \varepsilon) = ((p, \gamma, p') \in \Delta_0\ \mathcal{P})$
$|\ (p, \gamma) \hookrightarrow (p', [\gamma']) = ((p, \gamma, p', \gamma') \in \Delta_1\ \mathcal{P})$
$|\ (p, \gamma) \hookrightarrow (p', [\gamma_1, \gamma_2]) = ((p, \gamma, p', \gamma_1, \gamma_2) \in \Delta_2\ \mathcal{P})$
$|\ (p, \gamma) \hookrightarrow (p', \text{-}) = \mathit{False}$

lemma *PS-rule-mem-constraints*[*intro*, *dest*]:
assumes $(p, \gamma) \hookrightarrow (p', w)$
shows $p \in \mathit{Control}\ \mathcal{P}\ \gamma \in \Sigma\ \mathcal{P}\ p' \in \mathit{Control}\ \mathcal{P}\ \mathit{set}\ w \subseteq \Sigma\ \mathcal{P}$ **using** *assms* **and** *PS-axioms*
by (*cases rule*: *PS-rule.cases*[*of* $((p, \gamma), (p', w))$], *auto simp*: *PS-def*)+

A.3.1 Definition

definition *PS-transition-rel* :: $('c, 's)\ \mathit{config}\ \mathit{rel}$
where
PS-transition-rel $\equiv$
$\{((p, \gamma\ \#\ w), (p', w'\ @\ w))\ |\ p\ \gamma\ w\ p'\ w'.$
$\mathit{set}\ w \subseteq \Sigma\ \mathcal{P} \wedge \mathit{set}\ w' \subseteq \Sigma\ \mathcal{P} \wedge (p, \gamma) \hookrightarrow (p', w')\}$

abbreviation *PS-transition-frel* :: $('c, 's)\ \mathit{config} \Rightarrow ('c, 's)\ \mathit{config} \Rightarrow \mathit{bool}$
(**infix** $\Rightarrow$ *100*)
where
$c \Rightarrow c' \equiv (c, c') \in \mathit{PS\text{-}transition\text{-}rel}$

A.3.2 Properties

lemma *PS-transition-append*[*intro*]:
$(p, w) \Rightarrow (p', w') \Longrightarrow \mathit{set}\ u \subseteq \Sigma\ \mathcal{P} \Longrightarrow (p, w\ @\ u) \Rightarrow (p', w'\ @\ u)$
by (*auto simp*: *PS-transition-rel-def*)

lemma *PS-transition-rel-mem-constraints*[*intro*]:
assumes $(p, w) \Rightarrow (p', w')$
shows $p \in \mathit{Control}\ \mathcal{P}\ \mathit{set}\ w \subseteq \Sigma\ \mathcal{P}\ p' \in \mathit{Control}\ \mathcal{P}\ \mathit{set}\ w' \subseteq \Sigma\ \mathcal{P}$ **using** *assms*
by (*auto simp*: *PS-transition-rel-def*)

abbreviation *PS-transition-tr* :: $('c, 's)\ \mathit{config} \Rightarrow ('c, 's)\ \mathit{config} \Rightarrow \mathit{bool}$
(**infix** $\Rightarrow*$ *100*)
where $c \Rightarrow* c' \equiv (c, c') \in \mathit{PS\text{-}transition\text{-}rel}\ \hat{}*$

notation *PS-transition-tr* (**infixl** $\Rightarrow^*$ *100*)

lemma *rtrancl-f*[*intro*]:
assumes $\bigwedge a\ b.\ (a, b) \in r \Longrightarrow (f\ a, f\ b) \in r$
shows $(a, b) \in r\hat{}* \Longrightarrow (f\ a, f\ b) \in r\hat{}*$

apply (*induct rule*: *rtrancl-induct*)
using *assms* **by** *simp-all fastsimp*

lemma *rtrancl-f-2*[*intro*]:
assumes $\bigwedge a\ a'\ b\ b'.\ ((a,\ a'),\ (b,\ b')) \in r \Longrightarrow ((f\ a,\ f'\ a'),\ (f\ b,\ f'\ b')) \in r$
shows $((a,\ a'),\ (b,\ b')) \in r\hat{}* \Longrightarrow ((f\ a,\ f'\ a'),\ (f\ b,\ f'\ b')) \in r\hat{}*$
using *assms* **and** *rtrancl-f*[**where** $f = \lambda\ (a,\ a').\ (f\ a,\ f'\ a')$] **by** *auto*

lemma *PS-transition-tr-append*[*intro*]:
assumes *set* $u \subseteq \Sigma\ \mathcal{P}$
shows $(p,\ w) \Rightarrow* (p',\ w') \Longrightarrow (p,\ w\ @\ u) \Rightarrow* (p',\ w'\ @\ u)$
using *assms* **by** *auto*

lemma *PS-transition-tr-mem-constraints*[*intro*]:
assumes $(p,\ w) \Rightarrow* (p',\ w')$
and $p \in$ *Control* $\mathcal{P}$
shows $p' \in$ *Control* $\mathcal{P}$ **using** *assms*
by (*induct rule*: *rtrancl-induct2*) *auto*

lemma *PS-transition-tr-rev-mem-constraints*[*intro*]:
assumes $(p,\ w) \Rightarrow* (p',\ w')$
and $p' \in$ *Control* $\mathcal{P}$
shows $p \in$ *Control* $\mathcal{P}$ **using** *assms*
by (*induct rule*: *rtrancl-induct2*) *auto*

A.3.3 The set $pre^*\ C$

definition *prestar* :: ('*c*, '*s*) *config set* ⇒ ('*c*, '*s*) *config set*
(*pre∗* - [*101*] *100*)
where $pre*\ C = \{c.\ (\exists\, c' \in C.\ c \Rightarrow* c')\}$

notation *prestar* (*pre** *1000*)

definition *poststar* :: ('*c*, '*s*) *config set* ⇒ ('*c*, '*s*) *config set*
(*post∗* - [*101*] *100*)
where $post*\ C = \{c.\ (\exists\, c' \in C.\ c' \Rightarrow* c)\}$

notation *poststar* (*post** *1000*)

end

A.4 $\mathcal{P}$-Automata

$\mathcal{P}$-automata are word automata which recognize sets of configuations of the fixed pushdown system $\mathcal{P}$

record ('*c* :: *fresh*, '*s*) *PA-rec* =
Q :: '*c set*
Δ :: ('*c* ∗ '*s* ∗ '*c*) *set*
Final :: '*c set*

locale *PA* =
fixes $\mathcal{P}$ :: (*'c* :: *fresh*, *'s*) *PS-rec*
fixes $\mathcal{A}$ :: (*'c*, *'s*) *PA-rec*
assumes *PS* $\mathcal{P}$
assumes *finite* (*Q* $\mathcal{A}$)
assumes $\Delta\ \mathcal{A} \subseteq Q\ \mathcal{A} \times \Sigma\ \mathcal{P} \times Q\ \mathcal{A}$
assumes *Final* $\mathcal{A} \subseteq Q\ \mathcal{A}$
assumes *Control* $\mathcal{P} \subseteq Q\ \mathcal{A}$

lemma *PA-Control*[*intro*]: $[\![PA\ \mathcal{P}\ \mathcal{A};\ p \in Control\ \mathcal{P}]\!] \Longrightarrow p \in Q\ \mathcal{A}$ **by** (*auto simp*: *PA-def*)
lemma *PA-Final*[*intro*]: $[\![PA\ \mathcal{P}\ \mathcal{A};\ q \in Final\ \mathcal{A}]\!] \Longrightarrow q \in Q\ \mathcal{A}$ **by** (*auto simp*: *PA-def*)

A.4.1 Recognition and language

abbreviation *delta1* :: (*'c* :: *fresh*, *'s*) *PA-rec* ⇒ *'c* ⇒ *'s* ⇒ *'c* ⇒ *bool*
(- ⊢ - -→ - [*101*, *0*, *0*, *101*] *100*)
where $\mathcal{A} \vdash q\ -a\rightarrow\ q' \equiv (q,\ a,\ q') \in \Delta\ \mathcal{A}$

lemma *delta1-mem-constraints*[*intro*, *dest*]:
assumes *PA* $\mathcal{P}$ $\mathcal{A}$
and $\mathcal{A} \vdash q\ -a\rightarrow\ q'$
shows $q \in Q\ \mathcal{A}\ a \in \Sigma\ \mathcal{P}\ q' \in Q\ \mathcal{A}$
using *assms*
by (*auto simp*: *PA-def*)

inductive *delta* :: (*'c* :: *fresh*, *'s*) *PA-rec* ⇒ *'c* ⇒ *'s list* ⇒ *'c* ⇒ *bool*
where
base[*iff*]: $q \in Q\ \mathcal{A} \Longrightarrow delta\ \mathcal{A}\ q\ \varepsilon\ q$ |
step[*intro*]: $[\![delta1\ \mathcal{A}\ q\ a\ q';\ delta\ \mathcal{A}\ q'\ w\ q'']\!] \Longrightarrow delta\ \mathcal{A}\ q\ (a\#w)\ q''$

notation *delta* (- ⊢ - -→* - [*101*, *0*, *0*, *101*] *100*)

lemma *delta-mem-constraints*[*rule-format*, *intro*]:
$\mathcal{A} \vdash q\ -w\rightarrow*\ q' \Longrightarrow PA\ \mathcal{P}\ \mathcal{A} \longrightarrow (q \in Q\ \mathcal{A} \wedge q' \in Q\ \mathcal{A})$
by (*induct rule*: *delta.induct*) *auto*
lemma *delta-word-mem-constraints*[*rule-format*, *intro*]:
$\mathcal{A} \vdash q\ -w\rightarrow*\ q' \Longrightarrow PA\ \mathcal{P}\ \mathcal{A} \longrightarrow set\ w \subseteq \Sigma\ \mathcal{P}$
by (*induct rule*: *delta.induct*) *auto*

inductive-cases *delta-epsilon*[*elim*]: $\mathcal{A} \vdash q\ -\varepsilon\rightarrow*\ q'$
lemma *delta-epsilon-iff*[*iff*]: $(\mathcal{A} \vdash q\ -\varepsilon\rightarrow*\ q') = (q \in Q\ \mathcal{A} \wedge q = q')$
by *blast*
inductive-cases *delta-single*[*elim*]: $\mathcal{A} \vdash q\ -[a]\rightarrow*\ q'$
lemma *delta-single2*[*simp*]: $PA\ \mathcal{P}\ \mathcal{A} \Longrightarrow \mathcal{A} \vdash q\ -[a]\rightarrow*\ q' \longleftrightarrow \mathcal{A} \vdash q\ -a\rightarrow\ q'$

by *auto*
inductive-cases *delta-cons*[*elim*]: $\mathcal{A} \vdash q\ -a\#w\to*\ q'$
lemma *delta-cons-dest*[*dest*]: $\mathcal{A} \vdash q\ -a\#w\to*\ q'' \Longrightarrow (\exists\ q'.\ \mathcal{A} \vdash q\ -a\to\ q' \wedge \mathcal{A} \vdash q'\ -w\to*\ q'')$
by *blast*

lemma *delta-append*[*intro*]: $[\![\mathcal{A} \vdash q\ -u\to*\ q';\ \mathcal{A} \vdash q'\ -v\to*\ q'']\!] \Longrightarrow \mathcal{A} \vdash q\ -u@v\to*\ q''$
by (*induct u arbitrary*: *q*) (*auto*, *fast*)

lemma *delta-append-elim*[*elim*]:
$[\![\mathcal{A} \vdash q\ -u@v\to*\ q''; \bigwedge q'.\ [\![\mathcal{A} \vdash q\ -u\to*\ q';\ \mathcal{A} \vdash q'\ -v\to*\ q'']\!] \Longrightarrow P;\ PA\ \mathcal{P}\ \mathcal{A};\ q \in Q\ \mathcal{A}]\!] \Longrightarrow P$
proof (*induct u arbitrary*: *q P*)
fix q'
case (*Cons a u*)
then obtain *qa* **where** $\mathcal{A} \vdash q\ -a\to\ qa$ **and** $\mathcal{A} \vdash qa\ -u@v\to*\ q''$ **by** *auto*
with *Cons* **obtain** q' **where** $\mathcal{A} \vdash qa\ -u\to*\ q'$ **and** $\mathcal{A} \vdash q'\ -v\to*\ q''$ **by** *auto*
from ‹$\mathcal{A} \vdash q\ -a\to\ qa$› ‹$\mathcal{A} \vdash qa\ -u\to*\ q'$› **have** $\mathcal{A} \vdash q\ -a\#u\to*\ q'$ **..**
with ‹$\mathcal{A} \vdash q'\ -v\to*\ q''$› *Cons*(*3*) **show** *?case* **by** *auto*
qed *simp*

context *PS* **begin**

definition *language* :: ('*c* :: *fresh*, '*s*) *PA-rec* ⇒ ('*c*, '*s*) *config set* ($\mathcal{L}$)
where $\mathcal{L}\ \mathcal{A} \equiv \{(p, w).\ \exists\, q \in Final\ \mathcal{A}.\ p \in Control\ \mathcal{P} \wedge \mathcal{A} \vdash p\ -w\to*\ q\}$

lemma *language-intro*[*intro*]:
$[\![p \in Control\ \mathcal{P};\ \mathcal{A} \vdash p\ -w\to*\ q;\ q \in Final\ \mathcal{A}]\!] \Longrightarrow (p, w) \in \mathcal{L}\ \mathcal{A}$
by (*auto simp*: *language-def*)
lemma *language-elim*[*elim*]:
fixes $\mathcal{A}$
assumes $(p, w) \in \mathcal{L}\ \mathcal{A}$
obtains *q* **where** $p \in Control\ \mathcal{P}$ **and** $\mathcal{A} \vdash p\ -w\to*\ q$ **and** $q \in Final\ \mathcal{A}$
using *assms* **by** (*auto simp*: *language-def*)

definition *regular* :: ('*c*, '*s*) *config set* ⇒ *bool*
where *regular* (*C* :: ('*c*, '*s*) *config set*) ≡ ∃ $\mathcal{A}$ :: ('*c*, '*s*) *PA-rec*. *PA* $\mathcal{P}$ $\mathcal{A}$ ∧ *C* = $\mathcal{L}\ \mathcal{A}$

A.5 Saturation Procedure

This is the saturation procedure to transform an automaton $\mathcal{A}$ recognizing a language C into an automaton recognizing $pre^*\ C$: For evrery rule $(p, \gamma) \hookrightarrow (p', w)$, take q such that $\mathcal{A} \vdash p'\ -w\to*\ q$ and add a transition (p, γ, q) if it's not already present.

A.5.1 Definition

definition *pre-sat-rel* :: (*'c*, *'s*) *PA-rec rel*
  where *pre-sat-rel* = {(*A*, *A* (|Δ := *insert* (*p*, *γ*, *q*) (Δ *A*)|)) | *A p p' q γ w*.
    PA P A ∧ *p* ∈ *Control P* ∧ *γ* ∈ Σ *P* ∧ *q* ∈ *Q A* ∧ *p'* ∈ *Control P* ∧
    (*p*, *γ*) ↪ (*p'*, *w*) ∧
    A ⊢ *p'* −*w*→* *q* ∧
    ¬ *A* ⊢ *p* −*γ*→ *q*}

abbreviation *pre-sat-frel* :: (*'c*, *'s*) *PA-rec* ⇒ (*'c*, *'s*) *PA-rec* ⇒ *bool*
(**infix** ↦ *100*)
  where *A* ↦ *A'* ≡ (*A*, *A'*) ∈ *pre-sat-rel*

lemma *pre-sat-PA*[*intro*]:
  assumes *A* ↦ *A'*
  shows *PA P A* **and** *PA P A'*
  using *assms* **and** *PS-axioms*
  by (*auto simp*: *pre-sat-rel-def PA-def*)

abbreviation *pre-sat-tr* :: (*'c*, *'s*) *PA-rec* ⇒ (*'c*, *'s*) *PA-rec* ⇒ *bool* (**infix** ↦* *100*)
  where *A* ↦* *A'* ≡ (*A*, *A'*) ∈ *pre-sat-rel*^*

notation *pre-sat-tr* (**infix** ↦^* *100*)

A.5.2 Invariants

lemma *pre-sat-tr-PA*[*rule-format*, *dest*]: *A* ↦* *A'* ⟹ *PA P A* ⟶ *PA P A'*
  by (*induct rule*: *rtrancl-induct*) *auto*
lemma *pre-sat-Q*[*dest*]: *A* ↦ *A'* ⟹ *Q A* = *Q A'*
  by (*auto simp*: *pre-sat-rel-def*)
lemma *pre-sat-delta1*: ⟦*A* ⊢ *q* −*a*→ *q'*; *A* ↦ *A'*⟧ ⟹ *A'* ⊢ *q* −*a*→ *q'*
  by (*auto simp*: *pre-sat-rel-def*)
lemma *pre-sat-delta*: ⟦*A* ⊢ *q* −*w*→* *q'*; *A* ↦ *A'*⟧ ⟹ *A'* ⊢ *q* −*w*→* *q'*
  by (*induct rule*: *delta.induct*) (*auto elim*: *pre-sat-delta1*)
lemma *pre-sat-final*: *A* ↦ *A'* ⟹ *Final A* = *Final A'*
  by (*auto simp*: *pre-sat-rel-def*)

lemma *pre-sat-Delta*:
  assumes *A* ↦ *A'*
  obtains *p γ q* **where** Δ *A'* = *insert* (*p*, *γ*, *q*) (Δ *A*) ¬ *A* ⊢ *p* −*γ*→ *q* **using** *assms* **by** (*auto simp*: *pre-sat-rel-def*)

lemma *pre-sat-L*: *A* ↦ *A'* ⟹ *L A* ⊆ *L A'* **by** (*auto elim*: *pre-sat-delta simp*: *pre-sat-final language-def*)

lemma *pre-sat-tr-final*[*dest*]: *A* ↦* *A'* ⟹ *Final A* = *Final A'* **by** (*induct rule*: *rtrancl-induct*) (*auto dest*: *pre-sat-final*)
lemma *pre-sat-tr-L*[*dest*]: *A* ↦* *A'* ⟹ *L A* ⊆ *L A'* **apply** (*induct rule*: *rtrancl-induct*)
apply *simp* **using** *pre-sat-L* **by** *blast*

A.5.3 Saturated Automata

definition *sat*
  where *sat* $\mathcal{A} \equiv (PA\ \mathcal{P}\ \mathcal{A} \wedge \neg\ (\exists\ \mathcal{A}'.\ \mathcal{A} \mapsto \mathcal{A}'))$
lemma *sat-iff*: *sat* $\mathcal{A} = (PA\ \mathcal{P}\ \mathcal{A} \wedge ($
  $\forall\ p \in Control\ \mathcal{P}.$
  $\forall\ \gamma \in \Sigma\ \mathcal{P}.$
  $\forall\ p' \in Control\ \mathcal{P}.$
  $\forall\ q \in Q\ \mathcal{A}.$
  $\forall\, w.$
    $(p, \gamma) \hookrightarrow (p', w) \wedge \mathcal{A} \vdash p' -w\rightarrow* q \longrightarrow \mathcal{A} \vdash p -\gamma\rightarrow q))$ **unfolding** *sat-def pre-sat-rel-def* **by** *blast*

lemma *sat-iff-nomem*[*iff*]: $PA\ \mathcal{P}\ \mathcal{A} \Longrightarrow sat\ \mathcal{A} = (\forall\ p\ \gamma\ p'\ w.\ \forall\ q \in Q\ \mathcal{A}.\ (p, \gamma) \hookrightarrow (p', w) \wedge \mathcal{A} \vdash p' -w\rightarrow* q \longrightarrow \mathcal{A} \vdash p -\gamma\rightarrow q)$ **using** *sat-iff* **by** *blast*
— same lemma but without all implicit membership constraints

A.6 Certification of the saturation procedure

A.6.1 No Transition Should Lead to an Initial State

definition *init-no-income* :: $('c, 's)$ *PA-rec* $\Rightarrow$ *bool*
  where *init-no-income* $\mathcal{A} \equiv (\forall\ q \in Q\ \mathcal{A}.\ \forall\ \gamma \in \Sigma\ \mathcal{P}.\ \forall\ p \in Control\ \mathcal{P}.\ \neg\ \mathcal{A} \vdash q -\gamma\rightarrow p)$

lemma *init-no-income-elim*:
  assumes $\mathcal{A} \vdash q -\gamma\rightarrow p$
  $PA\ \mathcal{P}\ \mathcal{A}$
  $p \in Control\ \mathcal{P}$
  init-no-income $\mathcal{A}$
  shows P **using** *assms* **by** (*auto simp*: *init-no-income-def PA-def*)

We now show that this further assumption doesn't limit the power of the automata, given an automaton $\mathcal{A}$, we construct an equivalent automaton $\mathcal{A}'$ satisfying this condition

definition *fcopy* $\mathcal{A}$ = *fresh-copy* $(Q\ \mathcal{A})$ $(Control\ \mathcal{P})$
lemma *fsimp*:
  fixes q
  assumes $q \in Control\ \mathcal{P}$
  and *fcopy* $\mathcal{A}\ q \in Control\ \mathcal{P}$
  and $PA\ \mathcal{P}\ \mathcal{A}$
  shows *False*
proof −
  from *assms*(*1*,*2*) **have** *fcopy* $\mathcal{A}\ q \in$ *fresh-copy* $(Q\ \mathcal{A})$ $(Control\ \mathcal{P})$ ‘ $Control\ \mathcal{P} \cap (Control\ \mathcal{P} \cup Q\ \mathcal{A})$
    by (*auto simp*: *fcopy-def*)
  with *assms*(*3*) *finite-fresh-copy* **show** *False* **by** (*auto simp*: *PA-def PS-def*)
qed

definition *gcopy* $\mathcal{A}\ q = (if\ q \in Control\ \mathcal{P}\ then\ fcopy\ \mathcal{A}\ q\ else\ q)$

lemma *gsimp*:
fixes q
assumes $q \in Q\ \mathcal{A}$
and *gcopy* $\mathcal{A}\ q \in$ *Control* $\mathcal{P}$
and *PA* $\mathcal{P}\ \mathcal{A}$
shows *False* **using** *assms* **by** (*cases* $q \in$ *Control* $\mathcal{P}$) (*auto simp*: *gcopy-def dest*: *fsimp*)

definition *double-init* $\mathcal{A} = (\!|$
$Q = Q\ \mathcal{A} \cup$ *fcopy* $\mathcal{A}$ ‘ (*Control* $\mathcal{P}$),
$\Delta =$
$\{(p, \gamma,$ *gcopy* $\mathcal{A}\ q) \mid p\ \gamma\ q.\ (p, \gamma, q) \in \Delta\ \mathcal{A} \wedge p \in$ *Control* $\mathcal{P}\} \cup$
$\{($*gcopy* $\mathcal{A}\ q, \gamma,$ *gcopy* $\mathcal{A}\ q') \mid q\ \gamma\ q'.\ (q, \gamma, q') \in \Delta\ \mathcal{A}\}$,
Final $=$ *Final* $\mathcal{A} \cup$ *gcopy* $\mathcal{A}$ ‘ (*Final* $\mathcal{A}$)$|\!)$

lemma *double-init-Q-sub*:
$Q\ \mathcal{A} \subseteq Q$ (*double-init* $\mathcal{A}$) **by** (*auto simp*: *double-init-def*)
lemma *g-copy-Q*: $q \in Q\ \mathcal{A} \Longrightarrow$ *gcopy* $\mathcal{A}\ q \in Q$ (*double-init* $\mathcal{A}$)
by (*auto simp*: *double-init-def gcopy-def fcopy-def*)
lemma *inj-on-f*: *PA* $\mathcal{P}\ \mathcal{A} \Longrightarrow$ *inj-on* (*fcopy* $\mathcal{A}$) (*Control* $\mathcal{P}$)
by (*auto simp*: *finite-fresh-copy fcopy-def PA-def PS-def*)
lemma *fsimp2*:
fixes $\mathcal{A}\ p$
assumes *A*: *PA* $\mathcal{P}\ \mathcal{A}$
assumes *p*: $p \in$ *Control* $\mathcal{P}$
assumes *f*: *fcopy* $\mathcal{A}\ p \in Q\ \mathcal{A}$
shows *False*
proof −
from *p f* **have** *fcopy* $\mathcal{A}\ p \in$ *fcopy* $\mathcal{A}$ ‘ *Control* $\mathcal{P} \cap (Q\ \mathcal{A} \cup$ *Control* $\mathcal{P})$ **by** *simp*
with *finite-fresh-copy A* **show** *False* **by** (*auto simp*: *PA-def PS-def fcopy-def*)
qed
lemma *inj-on-g*: *PA* $\mathcal{P}\ \mathcal{A} \Longrightarrow$ *inj-on* (*gcopy* $\mathcal{A}$) ($Q\ \mathcal{A}$)
by (*auto simp*: *inj-on-def gcopy-def dest*: *inj-on-f fsimp2*)

lemma *PA-double-init*:
fixes $\mathcal{A}$
assumes *PA* $\mathcal{P}\ \mathcal{A}$
shows *PA* $\mathcal{P}$ (*double-init* $\mathcal{A}$) **using** *assms* **by** (*auto simp*: *double-init-def gcopy-def fcopy-def PA-def PS-def*)

lemma *double-init-no-income*:
fixes $\mathcal{A}$
assumes *PA* $\mathcal{P}\ \mathcal{A}$
assumes *double-init* $\mathcal{A} \vdash p\ -\gamma\rightarrow\ q$
shows $q \notin$ *Control* $\mathcal{P}$

using *assms*
by (*auto simp*: *double-init-def dest*: *gsimp*)

lemma *delta1-double-init*:
fixes $\mathcal{A}$
assumes $\mathcal{A} \vdash q\ -\gamma\rightarrow\ q'$
shows *double-init* $\mathcal{A} \vdash$ *gcopy* $\mathcal{A}\ q\ -\gamma\rightarrow$ *gcopy* $\mathcal{A}\ q'$
using *assms* **by** (*auto simp*: *double-init-def*)

lemma *delta-double-init*:
fixes $\mathcal{A}$
assumes *PA* $\mathcal{P}\ \mathcal{A}$
assumes $\mathcal{A} \vdash p\ -w\rightarrow*\ q$
shows *double-init* $\mathcal{A} \vdash$ *gcopy* $\mathcal{A}\ p\ -w\rightarrow*$ *gcopy* $\mathcal{A}\ q$
using *assms*(*2*)
proof (*induct rule*: *delta.induct*)
case (*base q* $\mathcal{A}$) **thus** *?case* **by** (*auto simp*: *g-copy-Q*)
next
case (*step* $\mathcal{A}\ q\ a\ q'\ w\ q''$)
with *delta1-double-init* **show** *?case* **by** *fast*
qed

lemma *delta-double-init-rev*:
fixes *A*
assumes *PA* $\mathcal{P}\ \mathcal{A}$
assumes *double-init* $\mathcal{A} \vdash$ *gcopy* $\mathcal{A}\ q\ -w\rightarrow*$ *gcopy* $\mathcal{A}\ q'$
assumes $q \in Q\ \mathcal{A}$ **and** $q' \in Q\ \mathcal{A}$ — probably a consequence of the other assumptions but I find it easier like this
shows $\mathcal{A} \vdash q\ -w\rightarrow*\ q'$
using *assms*(*2*,*3*,*4*)
proof (*induct w arbitrary*: *q*)
case *Nil*
hence *eq*: *gcopy* $\mathcal{A}\ q =$ *gcopy* $\mathcal{A}\ q'$ **by** *simp*
with ‹*PA* $\mathcal{P}\ \mathcal{A}$› ‹$q \in Q\ \mathcal{A}$› ‹$q' \in Q\ \mathcal{A}$› **have** $q = q'$ **using** *inj-on-g* **unfolding** *inj-on-def* **by** *simp*
thus $\mathcal{A} \vdash q\ -\varepsilon\rightarrow*\ q'$ **using** ‹$q \in Q\ \mathcal{A}$› **by** *blast*
next
case (*Cons* $\gamma\ u$)
then obtain *gq1* **where**
γ: *double-init* $\mathcal{A} \vdash$ *gcopy* $\mathcal{A}\ q\ -\gamma\rightarrow$ *gq1* **and**
u: *double-init* $\mathcal{A} \vdash$ *gq1* $-u\rightarrow*$ *gcopy* $\mathcal{A}\ q'$
by *blast*
from γ ‹$q \in Q\ \mathcal{A}$› ‹*PA* $\mathcal{P}\ \mathcal{A}$› **obtain** *q1* **where** [*simp*]: *gq1* = *gcopy* $\mathcal{A}$ *q1* **and** γA: $\mathcal{A} \vdash q\ -\gamma\rightarrow$ *q1* **unfolding** *double-init-def*
proof (*auto dest*: *gsimp*)
fix *qq* q'
assume *H* [*of* q', *simplified*]: $\bigwedge q1.\ [\![$*gcopy* $\mathcal{A}\ q' =$ *gcopy* $\mathcal{A}$ *q1*; $\mathcal{A} \vdash q\ -\gamma\rightarrow$ *q1*$]\!] \Longrightarrow$ *thesis*
and *q*: $q \in Q\ \mathcal{A}$ **and** *A*: *PA* $\mathcal{P}\ \mathcal{A}$

```
    and γ: A ⊢ qq −γ→ q'
    and eq: gcopy A q = gcopy A qq
    from eq A q γ inj-on-g have [simp]: q = qq unfolding inj-on-def by blast
    from H γ show thesis by simp
  qed
  hence [simp]: q1 ∈ Q A using ‹PA P A› by blast
  from u[simplified] Cons have A ⊢ q1 −u→* q' by simp
  with γA show ?case by auto
qed

lemma double-init-L-1:
  fixes A
  assumes A: PA P A
  assumes w: A ⊢ p −w→* q
  assumes p: p ∈ Control P
  assumes q: q ∈ Final A
  shows ∃ q' ∈ Final (double-init A). double-init A ⊢ p −w→* q'
proof (cases w)
  case Nil note [simp] = this
  from A q have q ∈ Q A by (auto simp: PA-def)
  hence q ∈ Q (double-init A) by (auto simp: double-init-def)
  with A w p q show ?thesis by (simp add: double-init-def)
next
  case (Cons γ u) note [simp] = this
  {
   from w obtain q1 where g: A ⊢ p −γ→ q1 and u: A ⊢ q1 −u→* q by auto
   from g have double-init A ⊢ p −γ→ gcopy A q1 using p unfolding double-init-def
      by (auto simp: gcopy-def fcopy-def)
    moreover
    from u delta-double-init A have double-init A ⊢ gcopy A q1 −u→* gcopy A
q by simp
    ultimately
    have double-init A ⊢ p −γ # u→* gcopy A q by fast
  }
  moreover
  from q have gcopy A q ∈ Final (double-init A) by (simp add: double-init-def)
  ultimately
  show ?thesis by auto
qed

lemma double-init-g-inv:
  fixes A q q'
  assumes A: PA P A
  assumes double-init A ⊢ gcopy A q −w→* q'
  assumes q: q ∈ Q A
  shows ∃ q'' ∈ Q A. q' = gcopy A q'' using assms(2,3)
proof (induct w arbitrary: q)
  case Nil
  hence q' = gcopy A q by simp
```

```
  with Nil(2) show ?case by blast
next
  case (Cons γ u)
  then obtain gq1 where
    γ: double-init 𝒜 ⊢ gcopy 𝒜 q −γ→ gq1 and
    u: double-init 𝒜 ⊢ gq1 −u→* q' by blast
   from γ obtain q1 where [simp]: gq1 = gcopy 𝒜 q1 q1 ∈ Q 𝒜 unfolding
double-init-def
    using A by auto
  from u[simplified] Cons show ?case by simp
qed

lemma double-init-L-2:
  fixes 𝒜
  assumes A: PA 𝒫 𝒜
  assumes w: double-init 𝒜 ⊢ p −w→* q
  assumes p: p ∈ Control 𝒫
  assumes q: q ∈ Final (double-init 𝒜)
  shows ∃ q' ∈ Final 𝒜. 𝒜 ⊢ p −w→* q'
proof (cases w)
  case Nil note [simp] = this
  from A w p have q2: q ∈ Control 𝒫 q ∈ Q 𝒜 by (auto simp: PA-def)
  with A q gsimp[of q 𝒜] have q ∈ Final 𝒜
  proof (cases q ∈ Final 𝒜)
    case True thus ?thesis by assumption
  next
    case False then obtain q' where q = gcopy 𝒜 q' q' ∈ Final 𝒜
      using q unfolding double-init-def by auto
    with w p A PA-Final have gcopy 𝒜 q' ∈ Control 𝒫 q' ∈ Q 𝒜 by auto
    hence False using A by (auto dest: gsimp)
    thus ?thesis by simp
  qed
  with w A show ?thesis by auto
next
  case (Cons γ u) note [simp] = this
  from w obtain gq1 where γ: double-init 𝒜 ⊢ p −γ→ gq1 and u: double-init
𝒜 ⊢ gq1 −u→* q
    by auto
  from γ A p gsimp obtain q1 where γA: 𝒜 ⊢ p −γ→ q1 and [simp]: gq1 =
gcopy 𝒜 q1
    unfolding double-init-def by auto
  hence [simp]: q1 ∈ Q 𝒜 using A by blast
  from double-init-g-inv[OF A u[simplified]] obtain q' where [simp]: q' ∈ Q 𝒜 q
= gcopy 𝒜 q' by auto
  from delta-double-init-rev[OF A u[simplified]] have uA: 𝒜 ⊢ q1 −u→* q' by
simp
  moreover
  have q' ∈ Final 𝒜
  proof −
```

```
    {
      assume q ∈ Final 𝒜
      hence q ∈ Q 𝒜 using A by auto
      with A have q' ∉ Control 𝒫 by (auto simp: gcopy-def dest: fsimp2)
      hence q = q' by (simp add: gcopy-def)
      hence q' ∈ Final 𝒜 using ‹q ∈ Final 𝒜› by simp
    }
    moreover
    {
      assume q ∈ gcopy 𝒜 ` Final 𝒜
      then obtain q'' where gcopy 𝒜 q' = gcopy 𝒜 q'' q'' ∈ Final 𝒜 by auto
      with A ‹q' ∈ Q 𝒜› inj-on-g have q' = q'' unfolding inj-on-def by blast
      with ‹q'' ∈ Final 𝒜› have q' ∈ Final 𝒜 by blast
    }
    ultimately show ?thesis using q unfolding double-init-def by fastsimp
  qed
  ultimately show ?thesis using γA by auto
qed

lemma init-no-income-wlog:
  fixes 𝒜
  assumes PA 𝒫 𝒜
  shows PA 𝒫 (double-init 𝒜) init-no-income (double-init 𝒜) ℒ 𝒜 = ℒ (double-init
𝒜)
proof −
  from PA-double-init assms show PA 𝒫 (double-init 𝒜) by simp
  from double-init-no-income assms show init-no-income (double-init 𝒜) unfold-
ing init-no-income-def by blast
  from double-init-L-1 double-init-L-2 assms show ℒ 𝒜 = ℒ (double-init 𝒜)
    unfolding language-def by auto
qed
```

A.6.2 Main Lemmas

First lemma

```
lemma Lemma1:
  assumes PA 𝒫 𝒜
  assumes 𝒜 ↦* 𝒜'
  assumes sat 𝒜'
  assumes (p', w) ⇒* (p, v)
  assumes (p, v) ∈ ℒ 𝒜
  shows ∃ q ∈ Final 𝒜'. 𝒜' ⊢ p' −w→* q
using assms(4)
proof (induct rule: converse-rtrancl-induct2)
  case refl
  from assms(2,5) show ?case by blast
next
  case (step p' w p'' u)
  show ?case
```

proof −
from *assms*(*1*,*2*) **have** [*simp*]: *PA* $\mathcal{P}$ $\mathcal{A}'$ **by** *auto*
with *step* (*1*) **have** [*simp*]: $p'' \in Q\ \mathcal{A}'$ **by** *blast*
from *step*(*1*) **obtain** γ *w1 u1* **where**
[*simp*]: $w = \gamma\ \#\ w1$ **and** [*simp*]: $u = u1\ @\ w1$
and [*simp*]: $(p', \gamma) \hookrightarrow (p'', u1)$
by (*auto simp*: *PS-transition-rel-def*)
from *step*(*3*) **obtain** *q* **where**
[*simp*]: $q \in Final\ \mathcal{A}'$ **and** $\mathcal{A}' \vdash p'' -u\rightarrow* q$
by *blast*
then obtain *q1* **where**
Split: $\mathcal{A}' \vdash p'' -u1\rightarrow* q1\ \mathcal{A}' \vdash q1 -w1\rightarrow* q$
by (*auto elim*: *delta-append-elim*[**where** $\mathcal{P} = \mathcal{P}$])
hence [*simp*]: $q1 \in Q\ \mathcal{A}'$
by (*auto simp*: *delta-mem-constraints*[**where** $\mathcal{P} = \mathcal{P}$])
from *Split*(*1*) *assms step* **have** $\mathcal{A}' \vdash p' -\gamma\rightarrow q1$
using *sat-iff-nomem*[*of* $\mathcal{A}'$]
by *fastsimp*

from *this Split*(*2*) **show** $\exists\, q \in Final\ \mathcal{A}'.\ \mathcal{A}' \vdash p' -w\rightarrow* q$ **by** *auto*
qed
qed

second lemma

special induction lemma

This lemma is used to do induction on the number of times a new rule is used

lemma *new-rule-induct*:
assumes [*simp*]: *PA* $\mathcal{P}$ $\mathcal{A}$ **and** *Next*: $\mathcal{A}' = \mathcal{A}(\!|\Delta := insert\ (p1,\ \gamma,\ q')\ (\Delta\ \mathcal{A})|\!)$
assumes *A*: $\mathcal{A}' \vdash p -w\rightarrow* q$ **and** [*simp*]: $p \in Control\ \mathcal{P}$
assumes *Base-case*: $\bigwedge p\ w.\ [\![p \in Control\ \mathcal{P};\ \mathcal{A} \vdash p -w\rightarrow* q]\!] \Longrightarrow P\ p\ w$
assumes *Step-case*:
$\bigwedge u\ v\ p.\ [\![$
$\mathcal{A} \vdash p -u\rightarrow* p1;$
$p \in Control\ \mathcal{P};$
$\mathcal{A}' \vdash q' -v\rightarrow* q;$
$\bigwedge p2\ w2.\ \mathcal{A} \vdash p2 -w2\rightarrow* q' \Longrightarrow p2 \in Control\ \mathcal{P} \Longrightarrow P\ p2\ (w2@v)$
$]\!] \Longrightarrow P\ p\ (u@\gamma\#v)$
shows *P p w*
proof −
{
fix *v*
have $\bigwedge qm\ u\ p.\ [\![p \in Control\ \mathcal{P};\ \mathcal{A} \vdash p -u\rightarrow* qm;\ \mathcal{A}' \vdash qm -v\rightarrow* q]\!] \Longrightarrow P\ p\ (u@v)$
proof (*induct v*)
case (*Nil qh u*) **with** *Base-case* **show** *?case* **by** *simp*
next
case (*Cons e v qm u*) **note** *IHP*=*this*

```
      then obtain qh where Split: 𝒜' ⊢ qm −e→ qh 𝒜' ⊢ qh −v→* q
        by (auto intro: delta1-mem-constraints[where 𝒫 = 𝒫])
      show ?case proof (cases (qm,e,qh) = (p1,γ,q'))
        case False
        with Split(1) Next have 𝒜 ⊢ qm −e→ qh by auto
        hence 𝒜 ⊢ qm −[e]→* qh by (simp add: delta-single2[where 𝒫 = 𝒫])
        with IHP(3) have 𝒜 ⊢ p −u@[e]→* qh by blast
        with IHP(1,2) Split have P p ((u@[e])@v) by blast
        thus ?thesis by simp
      next
        case True note CASE=this
        with Split IHP have 𝒜 ⊢ p −u→* p1 ∧ 𝒜' ⊢ q' −v→* q
          and ⋀uu. 𝒜 ⊢ p −uu→* q' ⟹ P p (uu@v) by auto
        with Step-case CASE Cons show ?thesis by auto
      qed
    qed
  } note C=this
  from A C[of p [] p w] show ?thesis by (auto simp: delta.base[of p 𝒜] PA-Control[of
𝒫 𝒜 p])
qed

lemma ex-monotonic: ⟦∃ x. P x; ⋀x. P x ⟹ P' x⟧ ⟹ ∃ x. P' x by blast

lemma Lemma2:
  assumes 𝒜 ↦* 𝒜'
  and init-no-income 𝒜
  and 𝒜' ⊢ p −w→* q
  and [simp]: PA 𝒫 𝒜
  and p ∈ Control 𝒫
  shows ∃ p' w'. (p, w) ⇒* (p', w') ∧ 𝒜 ⊢  p' −w'→* q ∧ (q ∈ Control 𝒫 ⟶
w' = ε) using assms
proof (induct arbitrary: p w q rule: rtrancl-induct)
  case base
  thus ?case
  proof (cases w = ε)
    from base(4)[simp] have [simp]: p ∈ Q 𝒜 by (simp add: PA-Control[where
𝒫 = 𝒫])
    case False
    hence [simp]: butlast w @ [last w] = w by simp
    from base(2) obtain q1 where 𝒜 ⊢ p −butlast w→* q1
      and 𝒜 ⊢ q1 −last w→ q
      using delta-append-elim[of 𝒜 p butlast w [last w] - - 𝒫]
      by auto
   with ‹init-no-income 𝒜› have ¬ q ∈ Control 𝒫 by (auto intro: init-no-income-elim)
      — I don't understand why but it has to be used as an intro rule...
    with ‹𝒜 ⊢ p −w→* q› show ?thesis by blast
  qed auto — case w = ε is trivial
next
  let ?P = λ p w q. ∃ p' w'.
```

```
    (p, w) ⇒* (p', w') ∧
    A ⊢ p' −w'→* q ∧
    (q ∈ Control P ⟶ w' = ε)
  case (step B A')
  from step(1) have [simp]: PA P B by auto
  from ‹B ↦ A'› obtain p1 γ q' p2 w2 where
    [simp]: p1 ∈ Control P γ ∈ Σ P q' ∈ Q B p2 ∈ Control P
      and [simp]: A' = B(|Δ := insert (p1, γ, q') (Δ B)|)
      and (p1, γ) ↪ (p2, w2)
      and Bw2: B ⊢ p2 −w2→* q' and A' ⊢ p1 −γ→ q'
    by (auto simp: pre-sat-rel-def)
  let ?A' = B(|Δ := insert (p1, γ, q') (Δ B)|)
  from step(2) have [simp]: PA P ?A' by auto
  with step show ?case
  proof (induct rule: new-rule-induct[of B ?A' p1 γ q' p w q], simp-all)
    fix u v p
    assume A'v: ?A' ⊢ q' −v→* q hence [simp]: set v ⊆ Σ P
      by (auto simp: delta-word-mem-constraints[of ?A' q' v q P])
    assume ⋀p w q. ⟦B ⊢ p −w→* q; p ∈ Control P⟧ ⟹ ?P p w q
      B ⊢ p −u→* p1
      p ∈ Control P
    hence Pu: ?P p u p1 by blast
     assume IH: ⋀p2 w2. ⟦B ⊢ p2 −w2→* q'; ⋀p w q. ⟦B ⊢ p −w→* q; p ∈
Control P⟧ ⟹ True;
      ?A' ⊢ p2 −w2 @ v→* q; p2 ∈ Control P⟧
      ⟹ ?P p2 (w2 @ v) q

    {
      from Pu have (p, u) ⇒* (p1, ε) by simp
      hence (p, u @ γ # v) ⇒* (p1, γ # v)
        using PS-transition-tr-append[of γ # v p u p1 ε]
        by simp
      moreover
      from ‹(p1, γ) ↪ (p2, w2)› have (p1, γ # v) ⇒ (p2, w2 @ v)
        by (auto simp: PS-transition-rel-def)
      ultimately have (p, u @ γ # v) ⇒* (p2, w2 @ v) by simp
    }
    moreover
    {
      from Bw2 ‹B ↦ A'› have ?A' ⊢ p2 −w2→* q'
        by (auto dest: pre-sat-delta)
      with A'v have ?A' ⊢ p2 −w2 @ v→* q
        by blast
      moreover
      have ⋀p w q. ⟦B ⊢ p −w→* q; p ∈ Control P⟧ ⟹ True by simp
      ultimately have ?P p2 (w2 @ v) q
        using IH Bw2
        by simp
    }
```

ultimately
show *?P p* (*u* @ *γ* # *v*) *q*
by (*auto elim*!: *ex-monotonic*)
qed
qed

A.6.3 Correctness

theorem *sat-correct*:
assumes *init-no-income* $\mathcal{A}$
and $\mathcal{A} \mapsto* \mathcal{A}'$
and *sat* $\mathcal{A}'$
and *PA* $\mathcal{P}$ $\mathcal{A}$
shows $\mathcal{L}\ \mathcal{A}' = pre*\ (\mathcal{L}\ \mathcal{A})$
proof *auto*
fix *p w*
assume (*p*,*w*) ∈ *pre*∗ (*L A*)
then obtain *p′ w′* **where**
(*p*, *w*) ⇒∗ (*p′*, *w′*) (*p′*, *w′*) ∈ *L A*
and [*simp*]: *p* ∈ *Control P*
by (*auto simp*: *prestar-def*)
thus (*p*, *w*) ∈ *L A′*
unfolding *language-def*
using *Lemma1* **and** *assms*
by *blast*
next
fix *p w*
from ‹*A* ↦∗ *A′*› **have** [*simp*]: *Final A′* = *Final A* **by** *blast*
assume (*p*, *w*) ∈ *L A′*
then obtain *q* **where**
A′ ⊢ *p* −*w*→∗ *q* **and** [*simp*]: *p* ∈ *Control P* **and** [*simp*]:*q* ∈ *Final A′*
unfolding *language-def* **by** *blast*
with *Lemma2 assms* **obtain** *p′ w′* **where** (*p*, *w*) ⇒∗ (*p′*, *w′*) *A* ⊢ *p′* −*w′*→∗ *q*
by *metis*
thus (*p*, *w*) ∈ *pre*∗ *L A*
unfolding *language-def prestar-def*
using ‹*q* ∈ *Final A′*›
by (*subgoal-tac p′* ∈ *Control P*, *auto*)
qed

A.6.4 Termination

lemma *wf-bounded-measure*:
fixes *ub* :: *′a* ⇒ *nat* **and** *f* :: *′a* ⇒ *nat*
assumes ⋀*a b*. (*b*,*a*) : *r* ⟹ *ub b* ≤ *ub a* ∧ *ub a* ≥ *f b* ∧ *f b* > *f a*
shows *wf r*
apply (*rule wf-subset*[*OF wf-measure*[*of* λ*a*. *ub a* − *f a*]])
apply (*auto dest*: *assms*)
done

lemma *wf-bounded-set*:
fixes *ub* :: *'a* ⇒ *'b set* **and** *f* :: *'a* ⇒ *'b set*
assumes ⋀*a b*. (*b*,*a*) : *r* ⟹
finite(*ub a*) ∧ *ub b* ⊆ *ub a* ∧ *ub a* ⊇ *f b* ∧ *f b* ⊃ *f a*
shows *wf r*
apply (*rule wf-bounded-measure*[*of r* λ*a*. *card*(*ub a*) λ*a*. *card*(*f a*)])
apply (*drule assms*)
apply (*blast intro*: *card-mono finite-subset psubset-card-mono dest*:
psubset-eq[*THEN iffD2*])
done

theorem *insert-new*: *x* ∉ *A* ⟹ *A* ≠ *insert x A* **by** *blast*

theorem *wf-sat-rel*: *wf* {(𝒜′, 𝒜). 𝒜 ↦ 𝒜′}
proof (*intro wf-bounded-set*[*of* - λ 𝒜. *Q* 𝒜 × Σ 𝒫 × *Q* 𝒜 Δ], *simp*)
fix 𝒜 𝒜′
assume 𝒜 ↦ 𝒜′
with *pre-sat-PA pre-sat-Q pre-sat-delta1 pre-sat-Delta insert-new* **show**
finite (*Q* 𝒜 × Σ 𝒫 × *Q* 𝒜) ∧
Q 𝒜′ × Σ 𝒫 × *Q* 𝒜′ ⊆ *Q* 𝒜 × Σ 𝒫 × *Q* 𝒜 ∧
Δ 𝒜′ ⊆ *Q* 𝒜 × Σ 𝒫 × *Q* 𝒜 ∧
Δ 𝒜 ⊂ Δ 𝒜′
unfolding *PA-def PS-def*
apply *auto*
by (*metis* ‹𝒜 ↦ 𝒜′› *in-measures*(*1*) *insert-new*)
qed
corollary *saturation*:
fixes 𝒜
assumes *PA* 𝒫 𝒜
shows ∃ 𝒜′. *PA* 𝒫 𝒜′ ∧ 𝒜 ↦* 𝒜′ ∧ *sat* 𝒜′
unfolding *sat-def* **using** *assms*
proof (*induct rule*: *wf-induct*[*OF wf-sat-rel*], *auto*)
fix *x*
assume *H0*: *PA* 𝒫 *x*
assume *H1*: ∀ *y*. *x* ↦ *y* ⟶ *PA* 𝒫 *y* ⟶ (∃ 𝒜′. *PA* 𝒫 𝒜′ ∧ *y* ↦* 𝒜′ ∧ *PA* 𝒫
𝒜′ ∧ (∀ *z*. ¬ 𝒜′ ↦ *z*))
show ∃ 𝒜′. *PA* 𝒫 𝒜′ ∧ *x* ↦* 𝒜′ ∧ *PA* 𝒫 𝒜′ ∧ (∀ *z*. ¬ 𝒜′ ↦ *z*)
proof (*cases* ∀ *z*. ¬ *x* ↦ *z*)
case *True* **with** *H0* **show** *?thesis* **by** *blast*
next
case *False*
then obtain *y* **where** *H3*: *x* ↦ *y* **by** *blast*
with *H0* **have** *PA* 𝒫 *y* **by** *auto*
from *H1*[*rule-format*, **where** *y* = *y*, *OF H3 this*] **obtain** *z*
where *PA* 𝒫 *z* **and** *y* ↦* *z* **and** ⋀*w*. ¬ *z* ↦ *w*
by *blast*
with *H3* **show** *?thesis* **by** *auto*
qed
qed

A.6.5 *regular* $C \Longrightarrow$ *regular* (*pre** C)

corollary *regular* $C \Longrightarrow$ *regular* (*pre*$*$ C)
proof −
assume *regular* C
then obtain $\mathcal{A}$ **where** *PA* $\mathcal{P}$ $\mathcal{A}$ **and** $\mathcal{L}$ $\mathcal{A}$ = C
unfolding *regular-def* **by** *blast*
then obtain $\mathcal{A}'$ **where** *PA* $\mathcal{P}$ $\mathcal{A}'$ **and** $\mathcal{L}$ $\mathcal{A}$ = $\mathcal{L}$ $\mathcal{A}'$ **and** *init-no-income* $\mathcal{A}'$
using *init-no-income-wlog* **by** *metis*
from ‹*PA* $\mathcal{P}$ $\mathcal{A}'$› **obtain** $\mathcal{A}''$ **where** $\mathcal{A}' \mapsto* \mathcal{A}''$ **and** *sat* $\mathcal{A}''$ *PA* $\mathcal{P}$ $\mathcal{A}''$
using *saturation* **by** *blast*
with ‹*PA* $\mathcal{P}$ $\mathcal{A}'$› **and** ‹*init-no-income* $\mathcal{A}'$› **and** *sat-correct* **have** $\mathcal{L}$ $\mathcal{A}''$ = *pre*$*$ ($\mathcal{L}$ $\mathcal{A}'$)
by *simp*
with ‹$\mathcal{L}$ $\mathcal{A}$ = C› **and** ‹$\mathcal{L}$ $\mathcal{A}$ = $\mathcal{L}$ $\mathcal{A}'$› **have** $\mathcal{L}$ $\mathcal{A}''$ = *pre*$*$ C
by *simp*
with ‹*PA* $\mathcal{P}$ $\mathcal{A}''$› **show** *regular* (*pre*$*$ C)
unfolding *regular-def* **by** *blast*
qed

end

end

References

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[2] S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped - a model-checker for pushdown systems. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.

[3] X. Leroy. Ocaml. http://caml.inria.fr/ocaml/.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. A proof assistant for higher-order logic, 2011.

[5] L. Paulson, T. Nipkow, and M. Wenzel. Isabelle. http://isabelle.in.tum.de/.

[6] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[7] M. Wenzel. The isabelle/isar reference manual, 2011.