Pure Prolog Execution in 21 Rules

M. Kulaš

FernUniversität Hagen, FB Informatik, D-58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract. A simple mathematical definition of the 4-port model for pure Prolog is given. The model combines the intuition of ports with a compact representation of execution state. Forward and backward derivation steps are possible. The model satisfies a modularity claim, making it suitable for formal reasoning.

1 Introduction

In order to formally handle (specify and prove) some properties of Prolog execution, we needed above all a definition of a port. A port is perhaps the single most popular notion in Prolog debugging, but theoretically it appears still rather elusive. The notion stems from the seminal article of L. Byrd [Byr80] which identifies four different types of control flow in a Prolog execution, as movements in and out of procedure *boxes* via the four *ports* of these boxes:

- *call*, entering the procedure in order to solve a goal,
- *exit*, leaving the procedure after a success, i. e. a solution for the goal is found,
- *fail*, leaving the procedure after the failure, i. e. there are no (more) solutions,
- *redo*, re-entering the procedure, i. e. another solution is sought for.

In this work, we present a formal definition of ports, which is a calculus of execution states, and hence provide a formal model of pure Prolog execution, S:PP. Our approach is to define ports by virtue of their effect, as *port transitions*. A port transition relates two *events*. An event is a state in the execution of a given query Q with respect to a given Prolog program Π. There are two restrictions we make:

1. the program Π has to be pure
2. the program Π shall first be transformed into a canonical form.

The first restriction concerns only the presentation in this paper, since our model has been prototypically extended to cover the control flow of full Standard Prolog, as given in [DEDC96]. The canonical form we use is the common single-clause representation. This representation is arguably 'near enough' to the original program, the only differences concern the head-unification (which is now delegated to the body) and the choices (which are now uniformly expressed as disjunction).

2 Preliminaries and the main idea

First we define the canonical form, into which the original program has to be transformed. Such a syntactic form appears as an intermediate stage in defining the Clark's completion of a logic program, and is used in logic program analysis. However, we are not aware of any consensus upon the name for this form. Some of the names in the literature are *single-clausal form* [Lin95] and *normalisation of a logic program* [KL02]. Here we use the name *canonical form*, partly on the grounds of our imposing a transformation on if-then as well (this additional transformation is of no interest in the present paper, which has to do only with pure Prolog, but we state it for completeness).

Definition 1 (canonical form of a predicate) We say that a predicate P/n is in the canonical form, if its definition consists of a single clause $P(X_1, ..., X_n)$:− $B; Bs$. Here B is a "canonical body", of the form $X_1{=}T_1, \ldots, X_n{=}T_n, G, Gs$, and $P(X_1, ..., X_n)$ is a "canonical head", i. e. $X_1, ..., X_n$ are distinct variables not appearing in G, Gs, $T_1, ..., T_n$. Further, Bs is a disjunction of canonical bodies (possibly empty), Gs is a conjunction of goals (possibly empty), and G is a goal (for facts: true). Additionally, each if-then goal $A \rightarrow B$ must be part of an if-then-else (like $A \rightarrow B; fail$). ■

Example 1 (canonical form) For the following program

```
q(a,b).
q(Z,c) :− r(Z).
r(c).
```

we obtain as canonical form

```
q(X,Y) :− X=a, Y=b, true; X=Z, Y=c, r(Z).
r(X) :− X=c, true.
```

❏

Having each predicate represented as one clause, and bearing in mind the box metaphor above, we identified some elementary execution steps. For simplicity we first disregard variables.

The following table should give some intuition about the idea. The symbols α, β in this table serve to identify the appropriate redo-transition, depending on the exit-transition. Transitions are deterministic, since the rules do not overlap.

Term	Port transitions in the context of Term			
$H{:-}B$	*call H* ⇾ *call B*	*exit B* ⇾ *exit H*	*fail B* ⇾ *fail H*	*redo H* ⇾ *redo B*
$A{,}B$	*call A,B* ⇾ *call A*	*exit A* ⇾ *call B*	*fail A* ⇾ *fail A,B*	*redo A,B* ⇾ *redo B*
		exit B ⇾ *exit A,B*	*fail B* ⇾ *redo A*	
$A;B$	*call A;B* ⇾ *call A*	*exit A* ⇾ $^{\alpha}$*exit A;B*	*fail A* ⇾ *call B*	$^{\alpha}$*redo A;B* ⇾ *redo A*
		exit B ⇾ $^{\beta}$*exit A;B*	*fail B* ⇾ *fail A;B*	$^{\beta}$*redo A;B* ⇾ *redo B*
true	*call* true ⇾ *exit* true			*redo* true ⇾ *fail* true
fail	*call* fail ⇾ *fail* fail			

Table 1. The idea of port transitions

Remark 1 (general goals) Observe that we extend the notion of a port, initially conceived for predicates, to *general goals*. The shifting of attention from predicates to goals is the key idea of this approach. □

Notation 1 (distinguishing meta-level from object-level) In the following we show object-level terms (i. e. actual Prolog terms) in sans serif, like true. Meta-level terms (i. e. anything else in the calculus) will be shown in italics, like *call*,Σ, or in blackboard font, like $\mathbb{U}$,$\mathbb{\Sigma}$. □

Each transition pertains to a certain context, as indicated in Table 1. In the next step towards the new definition of ports we shall make this dependency explicit, by adding a parameter to each event.

Example 2 (good, bad and main) Relative to the program

```
main :- good, bad.
good.
```

there are the following execution steps for main:

```
call main ->
  call (good, bad) ->
    call good ->
      call true ->
      exit true ->
    exit good ->
    call bad ->
    fail bad ->
    redo good ->
      redo true ->
      fail true ->
    fail good ->
  fail (good, bad) ->
fail main
```

The indentations should suggest the context of the transitions, which is not very satisfying, since we want our representation to be entirely symbolic, and therefore visual aspects may not be part of the definition. So we provide the context information within the calculus, by means of a *stack of ancestors*, or A-stack. Hereby we define the immediate ancestor (the parent) of a goal to be the context of the transition. On some reflection, this is not enough. In case of a redo of an atomary goal, like *redo* true above, we need to know how the goal was resolved, in order to see the remaining alternatives. Since it is possible, in full Prolog, that a predicate definition changes between an exit and a redo, simply accessing the program would not guarantee the retrieval of the definition effective at the time of call. For this reason we memorize, at an exit of an atomary goal, the effectively used definition (more about this on page 6). Also, on exit from a disjunction, some kind of memoing of the used disjunct is necessary. So we tried combining the memoing (both kinds of memos: used definitions and used disjuncts) with the administration of variable bindings, into one *stack of bets*, or B-stack. One claim of this paper is that an A-stack and a B-stack are sufficient to represent the execution of pure Prolog. As an illustration of the two-stack idea, let us show the above derivation in complete detail. In Appendix B an example with variables is given. Each stack is enclosed in parentheses, $\bullet$ separates the elements, and *nil* marks the bottom of a stack.

call main , $\{\mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ call (good, bad) , $\{\text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ call good , $\{1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ call true , $\{\text{good} \bullet 1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ exit true , $\{\text{good} \bullet 1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ exit good , $\{1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{BY(\text{true, good}) \bullet \mathit{nil}\}$
$\rightarrow$ call bad , $\{2/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{BY(\text{true, good}) \bullet \mathit{nil}\}$
$\rightarrow$ fail bad , $\{2/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{BY(\text{true, good}) \bullet \mathit{nil}\}$
$\rightarrow$ redo good , $\{1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{BY(\text{true, good}) \bullet \mathit{nil}\}$
$\rightarrow$ redo true , $\{\text{good} \bullet 1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ fail true , $\{\text{good} \bullet 1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ fail good , $\{1/\text{good, bad} \bullet \text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ fail (good, bad) , $\{\text{main} \bullet \mathit{nil}\}$, $\{\mathit{nil}\}$
$\rightarrow$ fail main , $\{\mathit{nil}\}$, $\{\mathit{nil}\}$

❑

3 The calculus S:PP

We consider pure Prolog programs as given in Fig. 1, syntax domain "program", under restriction that every "definition" has to be in the canonical form.

Definition 2 (event) An *event* is a quadruple $(Port, Goal, A\text{-}stack, B\text{-}stack)$, as given by the grammar in Fig. 1, syntax domain "event". ■

Intuitively, an event is a state of Prolog execution, determined by four parameters:

- port
- current goal
- history of current goal (stack of generalized ancestors, for short: *A-stack)*
- current environment (stack of generalized bindings, *bets*, for short: *B-stack)*

Definition 3 (transition rule) Let Π be a program. Port transition rules wrt Π are listed in Fig. 2. ■

$$\begin{array}{lll}
\textbf{event} & ::= & \textbf{port goal} \left\langle \frac{\textbf{stack of bets}}{\textbf{stack of ancestors}} \right\rangle \\
\textbf{event} & ::= & \textbf{port goal}\,\textbf{,}\,\{\textbf{stack of ancestors}\}\textbf{,}\,\{\textbf{stack of bets}\}\ \%\ \text{inline} \\
\text{definition} & ::= & \text{atom :- goal} \\
\text{program} & ::= & \{\text{definition.}\}^{+} \\
\text{port} & ::= & \mathit{call} \mid \mathit{exit} \mid \mathit{fail} \mid \mathit{redo} \\
\text{goal} & ::= & \mathsf{true} \mid \mathsf{fail} \mid \text{atom} \mid \text{term=term} \mid \text{goal;goal} \mid \text{goal,goal} \\
\text{ancestor} & ::= & \mathsf{true} \mid \mathsf{fail} \mid \text{atom} \mid \text{term=term} \mid \text{tag/goal;goal} \mid \text{tag/goal,goal} \\
\text{tag} & ::= & \mathsf{1} \mid \mathsf{2} \\
\text{memo} & ::= & BY(\text{goal}, \text{atom}) \mid OR(\text{goal}, (\text{tag/goal;goal})) \\
\text{bet} & ::= & \text{mgu} \mid \text{memo} \\
\text{stack of Xs} & ::= & \mathit{nil} \mid \text{X} \bullet \text{stack of Xs}
\end{array}$$

Variables

$$\begin{array}{lll}
\mathbb{U}, \mathbb{V} & : \text{stack of ancestors}, & U : \text{ancestor} \\
\mathbb{\Sigma}, \mathbb{\Delta} & : \text{stack of bets}, & \Sigma : \text{bet} \\
\sigma & : \text{substitution} & \\
A, B, C, \ G, H & : \text{goal} & \\
G_A & : \text{atom} & \\
T & : \text{term} &
\end{array}$$

Semantic functions

$$\begin{array}{lll}
T_1 = T_2 & := & T_1 \text{ and } T_2 \text{ are identical} \\
\sigma(T) & = & \text{application of } \sigma \text{ upon } T \\
\text{mgu}(T_1, T_2) & = & \text{mgu of } T_1 \text{ and } T_2 \\
\text{substOf}(\mathbb{\Sigma}) & = & \text{current substitution } = \text{ composition of all mgus from } \mathbb{\Sigma}
\end{array}$$

$$\text{substOf}(\mathit{nil})(T) := T$$

$$\text{substOf}(\Sigma \bullet \mathbb{\Sigma})(T) := \begin{cases} \Sigma(\text{substOf}(\mathbb{\Sigma})(T)), & \text{if } \Sigma \text{ is an mgu} \\ \text{substOf}(\mathbb{\Sigma})(T), & \text{if } \Sigma \text{ is a memo} \end{cases}$$

Syntactic domains that we do not redefine, but take in their usual sense:
term (taken in the Prolog sense, as a superset of goal);
atom (atomary goal in logic programming);
substitution, mgu.

Fig. 1. Language of events

Conjunction

$$\mathit{call}\ A, B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{call}\ A \,\langle \tfrac{\Sigma}{1/A,B \,\bullet\, \mathbb{U}} \rangle \tag{S:conj:1}$$

$$\mathit{exit}\ A' \,\langle \tfrac{\Sigma}{1/A,B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{call}\ B'' \,\langle \tfrac{\Sigma}{2/A,B \,\bullet\, \mathbb{U}} \rangle,\ \text{with } B'' := \mathrm{substOf}(\Sigma)(B) \tag{S:conj:2}$$

$$\mathit{fail}\ A' \,\langle \tfrac{\Sigma}{1/A,B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ A, B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:conj:3}$$

$$\mathit{exit}\ B' \,\langle \tfrac{\Sigma}{2/A,B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{exit}\ A, B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:conj:4}$$

$$\mathit{fail}\ B' \,\langle \tfrac{\Sigma}{2/A,B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{redo}\ A \,\langle \tfrac{\Sigma}{1/A,B \,\bullet\, \mathbb{U}} \rangle \tag{S:conj:5}$$

$$\mathit{redo}\ A, B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{redo}\ B \,\langle \tfrac{\Sigma}{2/A,B \,\bullet\, \mathbb{U}} \rangle \tag{S:conj:6}$$

Disjunction

$$\mathit{call}\ A; B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{call}\ A \,\langle \tfrac{\Sigma}{1/A;B \,\bullet\, \mathbb{U}} \rangle \tag{S:disj:1}$$

$$\mathit{fail}\ A \,\langle \tfrac{\Sigma}{1/A;B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{call}\ B \,\langle \tfrac{\Sigma}{2/A;B \,\bullet\, \mathbb{U}} \rangle \tag{S:disj:2}$$

$$\mathit{fail}\ B \,\langle \tfrac{\Sigma}{2/A;B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ A; B \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:disj:3}$$

$$\mathit{exit}\ A \,\langle \tfrac{\Sigma}{1/A;B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{exit}\ A; B \,\langle \tfrac{OR(A,(1/A;B)) \,\bullet\, \Sigma}{\mathbb{U}} \rangle \tag{S:disj:4}$$

$$\mathit{exit}\ B \,\langle \tfrac{\Sigma}{2/A;B \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{exit}\ A; B \,\langle \tfrac{OR(B,(2/A;B)) \,\bullet\, \Sigma}{\mathbb{U}} \rangle \tag{S:disj:5}$$

$$\mathit{redo}\ A; B \,\langle \tfrac{OR(C,(N/A;B)) \,\bullet\, \Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{redo}\ C \,\langle \tfrac{\Sigma}{N/A;B \,\bullet\, \mathbb{U}} \rangle \tag{S:disj:6}$$

True

$$\mathit{call}\ \mathsf{true} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{exit}\ \mathsf{true} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:true:1}$$

$$\mathit{redo}\ \mathsf{true} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ \mathsf{true} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:true:2}$$

Fail

$$\mathit{call}\ \mathsf{fail} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ \mathsf{fail} \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:fail}$$

Explicit unification

$$\mathit{call}\ T_1 = T_2 \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \begin{cases} \mathit{exit}\ T_1 = T_2 \,\langle \tfrac{\sigma \,\bullet\, \Sigma}{\mathbb{U}} \rangle, & \text{if } \mathrm{mgu}(T_1, T_2) = \sigma \\ \mathit{fail}\ T_1 = T_2 \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle, & \text{otherwise} \end{cases} \tag{S:unif:1}$$

$$\mathit{redo}\ T_1 = T_2 \,\langle \tfrac{\sigma \,\bullet\, \Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ T_1 = T_2 \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:unif:2}$$

User-defined atomary goal G_A

$$\mathit{call}\ G_A \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \rightharpoonup \begin{cases} \mathit{call}\ \sigma(B) \,\langle \tfrac{\Sigma}{G_A \,\bullet\, \mathbb{U}} \rangle, & \text{if } H \mathrel{:-} B \text{ is a fresh renaming of a} \\ \quad \text{clause in } \Pi, \text{ and } \mathrm{mgu}(G_A, H) = \sigma, \text{ and } \sigma(G_A) = G_A & \\ \mathit{fail}\ G_A \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle, & \text{otherwise} \end{cases} \tag{S:atom:1}$$

$$\mathit{exit}\ B \,\langle \tfrac{\Sigma}{G_A \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{exit}\ G_A \,\langle \tfrac{BY(B, G_A) \,\bullet\, \Sigma}{\mathbb{U}} \rangle \tag{S:atom:2}$$

$$\mathit{fail}\ B \,\langle \tfrac{\Sigma}{G_A \,\bullet\, \mathbb{U}} \rangle \rightharpoonup \mathit{fail}\ G_A \,\langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:atom:3}$$

$$\mathit{redo}\ G_A \,\langle \tfrac{BY(B, G_A') \,\bullet\, \Sigma}{\mathbb{U}} \rangle \rightharpoonup \mathit{redo}\ B \,\langle \tfrac{\Sigma}{G_A' \,\bullet\, \mathbb{U}} \rangle \tag{S:atom:4}$$

Fig. 2. Operational semantics S:PP of pure Prolog

3.1 Remarks on the calculus

About event:

- *Current goal* is a generalization of *selected literal*: rather than focusing upon single literals, we focus upon goals.
- *Ancestor* of a goal is defined in a disambiguating manner, via *tags*.
- The notion of *environment* is generalized, to contain following *bets*:
  1. variable bindings,
  2. choices taken (OR-branches),
  3. used predicate definitions.

  Environment is represented by one stack, storing each bet as soon as it is computed. For an event to represent the state of pure Prolog execution, suffices here one environment and one ancestor stack.

About transitions:

- Port transition relation is functional. The same holds for its converse, if restricted on *legal events*, i. e. events that can be reached from an *initial event* of the form *call* $G \langle \frac{nil}{nil} \rangle$.
- This uniqueness of legal derivations enables *forward and backward* derivation steps, in the spirit of the Byrd's article.
- *Modularity* of derivation: The execution of a goal can be abstracted like for example *call* $G \langle \frac{\Sigma}{\mathbb{U}} \rangle \overset{*}{\rightarrow}$ *exit* $G \langle \frac{\Delta + \Sigma}{\mathbb{U}} \rangle$. Notice the same A-stack.

Remark 2 (atomary goal) By *atom* or *atomary goal* we denote only user-defined predications. So true, fail or $T_1 = T_2$ shall not be considered atoms. □

Remark 3 (mgu) The most general unifiers σ shall be chosen to be idempotent, i. e. $\sigma(\sigma(T)) = \sigma(T)$. □

Remark 4 (tags) The names A' or B' of (S:conj:2)–(S:conj:5) should only suggest that the argument is related to A or B, but the actual retrieval is determined by the tags 1 and 2, saying that respectively the first or the second conjunct are currently being tried. For example, the rule (S:conj:1) states that the call of A, B leads to the call of A with immediate ancestor $1/A, B$. This kind of add-on mechanism is necessary to be able to correctly handle a query like A, A where retrieval by unification would get stuck on the first conjunct. □

Remark 5 (canonical form) Note the requirement $\sigma(G_A) = G_A$ in (S:atom:1). Since the clauses are in canonical form, unifying the head of a clause with a goal could do no more than rename the goal. Since we do not need a renaming of the goal, we may fix the mgu to just operate on the clause. □

Remark 6 (logical update view) Observe how (S:atom:2) and (S:atom:4) serve to implement the *logical update view* of Lindholm and O'Keefe [LO87], saying that the definition of a predicate shall be fixed at the time of its call. This is further explained in the following remark. □

Remark 7 ("lazy" binding) Although we memorize the used predicate definition *on exit*, the definition will be unaffected by exit bindings, because *bindings are applied lazily*: Instead of "eagerly" applying any bindings as they occur (e. g. in $T_1 = T_2$, in resolution or in read), we chose to do this only in conjunction (in rule

(S:conj:2)) and nowhere else. Due to the rules (S:conj:1) and (S:conj:4), the exit bindings shall not affect the predicate definition like e. g. $\mathsf{p(X)} \coloneq \mathsf{q(X), r(X)}$.

Also, lazy bindings enable less 'jumpy' trace. A jumpy trace can be illustrated by the following exit event (assuming we applied bindings eagerly):

$$\mathit{exit}\ \mathsf{append([O], B, [O|B])}\ ,\ \{2/([\mathsf{I|B}] = [\mathsf{I|B}]), \mathsf{append([], B, B)} \bullet \mathbb{U}\},\ \Sigma$$

The problem consists in exiting the goal $\mathsf{append([], B, B)}$ via $\mathsf{append([O], B, [O|B])}$, the latter of course being no instance of the former. By means of lazy binding, we avoid the jumpiness, and at the same time make memoing definitions on exit possible. To ensure that the trace of a query execution shows the correct bindings, an event shall be printed only after the current substitution has been applied to it.

A perhaps more important collateral advantage of lazy binding is that a successful derivation (see Definition 11) can always be abstracted as follows:

$$\mathit{call\ Goal} \stackrel{*}{\rightarrow} \mathit{exit\ Goal}$$

even if *Goal* happened to get further instantiated in the course of this derivation. The instantiation will be reflected in the B-stack but not in the goal itself. □

4 Modelling Prolog execution

Definition 4 (port transition relation, converse) Let Π be a program. *Port transition relation* $\rightarrow$ wrt Π is defined in Fig. 2. The converse relation shall be denoted by $\leftarrow$. If $E_1 \rightarrow E$, we say that E_1 *leads to* E. An event E can be *entered*, if some event leads to it. An event E can be *left*, if it leads to some event. ■

Lemma 1 The relation $\rightarrow$ is functional, i. e. for each event E there can be at most one event E_1 such that $E \rightarrow E_1$. ■

Proof: The premisses of the transition rules are mutually disjunct, i. e. there are no critical pairs. □

Example 3 (converse relation) The converse of the port transition relation is not functional, since there may be more than one event leading to the same event:

$$\mathit{call}\ T_1 = T_2\ \langle\tfrac{nil}{nil}\rangle \rightarrow \mathit{fail}\ T_1 = T_2\ \langle\tfrac{nil}{nil}\rangle$$
$$\mathit{redo}\ T_1 = T_2\ \langle\tfrac{\sigma \bullet nil}{nil}\rangle \rightarrow \mathit{fail}\ T_1 = T_2\ \langle\tfrac{nil}{nil}\rangle$$

We could have prevented the ambiguous situation above and made converse relation functional as well, by giving natural conditions on redo-transitions for atomary goal and unification. However, further down it will be shown that, for events that are *legal*, the converse relation is functional anyway. ❑

Definition 5 (derivation) Let Π be a program. Let E_0, E be events. A Π-*derivation of* E *from* E_0, written as $E_0 \stackrel{*}{\rightarrow} E$, is a path from E_0 to E in the port transition relation wrt Π. We say that E can be *reached* from E_0. ■

Definition 6 (initial event, top-level goal) An *initial event* is any event of the form $\mathit{call}\ Q\ \langle\tfrac{nil}{nil}\rangle$, where Q is a goal. The goal Q of an initial event is called a *top-level goal*, or a *query*. ■

Definition 7 (legal derivation, legal event, execution) Let Π be a program. If there is a goal Q such that

$$call\ Q\,\langle\tfrac{nil}{nil}\rangle \overset{*}{\twoheadrightarrow} E_0 \overset{*}{\twoheadrightarrow} E$$

is a Π-derivation, then we say that $E_0 \overset{*}{\twoheadrightarrow} E$ is a *legal Π-derivation*, E is a *legal Π-event*, and $call\ Q\,\langle\tfrac{nil}{nil}\rangle \overset{*}{\twoheadrightarrow} E_0$ is a *Π-execution* of the query Q. ■

Definition 8 (final event) A legal event E is a *final* event wrt program Π, if there is no transition $E \twoheadrightarrow E_1$ wrt Π. ■

Definition 9 (parent of goal) If $E = Port\ G\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle$ is an event, and $\mathbb{U} = P \bullet \mathbb{V}$, then we say that P is the *parent* of G. ■

Notation 2 (selector tags) Function $\mathrm{Sel}(U)$ is defined as follows:

$$\mathrm{Sel}((1/A,B)) := A,\ \mathrm{Sel}((2/A,B)) := B$$

and analogously for disjunction. □

Definition 10 (push/pop event) Let E be an event with the port $Port$. If $Port$ is one of $call, redo$, then E is a *push* event. If $Port$ is one of $exit, fail$, then E is a *pop* event. ■

Lemma 2 (final event) If E is a legal pop event, and its A-stack is not empty, then $\exists E_1 : E \twoheadrightarrow E_1$ ■

Proof (sketch): According to the rules (see also Appendix A), the possibilities to leave an exit event are:

$$\begin{aligned}
exit\ A'\,\langle\tfrac{\Sigma}{1/A,B\,\bullet\,\mathbb{U}}\rangle &\twoheadrightarrow call\ B''\,\langle\tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}}\rangle,\ \text{with } B'' := \mathrm{substOf}(\Sigma)(B)\\
exit\ B'\,\langle\tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}}\rangle &\twoheadrightarrow exit\ A,B\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle\\
exit\ A\,\langle\tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}}\rangle &\twoheadrightarrow exit\ A;B\,\langle\tfrac{OR(A,(1/A;B))\,\bullet\,\Sigma}{\mathbb{U}}\rangle\\
exit\ B\,\langle\tfrac{\Sigma}{2/A;B\,\bullet\,\mathbb{U}}\rangle &\twoheadrightarrow exit\ A;B\,\langle\tfrac{OR(B,(2/A;B))\,\bullet\,\Sigma}{\mathbb{U}}\rangle\\
exit\ B\,\langle\tfrac{\Sigma}{G_A\,\bullet\,\mathbb{U}}\rangle &\twoheadrightarrow exit\ G_A\,\langle\tfrac{BY(B,G_A)\,\bullet\,\Sigma}{\mathbb{U}}\rangle
\end{aligned}$$

These rules state that it is always possible to leave an exit event $exit\ G\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle$, save for the following two restrictions: The parent goal may not be true, fail or a unification; and if the parent goal P is a disjunction, then there has to hold

$$G = \mathrm{Sel}(P) \tag{1}$$

i. e. it is not possible to leave an event $exit\ A'\,\langle\tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}}\rangle$ if $A' \neq A$ (and similarly for the second disjunct). The first restriction is void, since a parent cannot be true, fail or a unification anyway, according to the rules. It remains to show that the second restriction is also void, i. e. a legal exit event has necessarily the property (1). Looking at the rules for entering an exit event, we note that the goal part of an exit event either comes from the A-stack, or is true or $T_1 = T_2$. The latter two possibilities we may exclude, because $exit\,\mathsf{true}\,\langle\tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}}\rangle$ can only be derived from $call\,\mathsf{true}\,\langle\tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}}\rangle$, which cannot be reached if $\mathsf{true} \neq A$. Similarly for unification.

So the goal part of a legal exit event must come from the A-stack. The elements of the A-stack originate from call/redo events, and they have the property (1). In conclusion, we can always leave a legal exit event with a nonempty A-stack. Similarly for a fail event. $\square$

Proposition 1 (uniqueness) If E is a legal event, then E can have only one legal predecessor, and only one successor. In case E is non-initial, there is exactly one legal predecessor. In case E is non-final, there is exactly one successor. $\blacksquare$

PROOF: The successor part follows from the functionality of $\rightarrowtriangle$. Looking at the rules, we note that only two kinds of events may have more than one predecessor: $\mathit{fail}\ G_A\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ and $\mathit{fail}\ T_1 = T_2\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$. Let $\mathit{fail}\ T_1 = T_2\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ be a legal event. Its predecessor may have been $\mathit{call}\ T_1 = T_2\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$, on the condition that T_1 and T_2 have no mgu (rule (S:unif:1)), or it could have been $\mathit{redo}\ T_1 = T_2\,\langle\frac{\sigma\bullet\Sigma}{\mathbb{U}}\rangle$ (rule (S:unif:2)). In the latter case, $\mathit{redo}\ T_1 = T_2\,\langle\frac{\sigma\bullet\Sigma}{\mathbb{U}}\rangle$ must be a legal event, so the B-stack $\sigma\bullet\Sigma$ had to be derived. The only rule able to derive such a B-stack is (S:unif:1), on the condition that the previous event was $\mathit{call}\ T_1 = T_2\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ and $\mathrm{mgu}(T_1, T_2) = \sigma$. Hence, there can be only one legal predecessor of $\mathit{fail}\ T_1 = T_2\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$, depending solely on T_1 and T_2. By a similar argument we can prove that $\mathit{fail}\ G_A\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ can have only one legal predecessor. This concludes the proof of functionality of the converse relation, if restricted to the set of legal events. $\square$

NOTATION 3 (IMPOSSIBLE EVENT) As a notational convenience, all the events which are not final and do not lead to any further events by means of transitions with respect to the given program, are said to lead to the *impossible event*, written as $\bot$. Analogously for events that are not initial events and cannot be entered. In particular, $\mathit{redo}\,\mathsf{fail} \rightarrowtriangle \bot$ and $\mathit{exit}\,\mathsf{fail} \leftarrowtriangle \bot$ with respect to any program. Some impossible events are: $\mathit{call}\ G\,\langle\frac{\sigma\bullet\mathit{nil}}{\mathit{nil}}\rangle$, $\mathit{redo}\ G\,\langle\frac{\Sigma}{\mathit{nil}}\rangle$ (cannot be entered, non-initial), and $\mathit{redo}\,\mathsf{p}\,\langle\frac{\mathit{nil}}{\mathbb{U}}\rangle$ (cannot be left, non-final). $\square$

Lemma 3 (non-legal event) If $E \overset{*}{\rightarrowtriangle} \bot$, then E is not legal. If $E \overset{*}{\leftarrowtriangle} \bot$, then E is not legal. $\blacksquare$

PROOF: Let $E \leftarrowtriangle E_1$. If E is legal, then, because of the uniqueness of the transition, E_1 has to be legal as well. $\square$

Lemma 4 (call is up-to-date) For a legal call event $\mathit{call}\ G\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ holds that $G = \mathrm{substOf}(\Sigma)(G)$, meaning that the substitutions from the B-stack are *already applied* upon the goal to be called. In other words, the goal of any legal call event is up-to-date relative to the current substitution. $\blacksquare$

Notice that this property holds only for call events.

NOTATION 4 (STACK CONCATENATION) Concatenation of stacks we denote by $+$. Concatenating to both stacks of an event we denote by $\ddagger$: If $E = \mathit{Port}\ G\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$, then $E \ddagger \langle\frac{\mathbb{A}}{\mathbb{V}}\rangle := \mathit{Port}\ G\,\langle\frac{\Sigma + \mathbb{A}}{\mathbb{U} + \mathbb{V}}\rangle$. $\square$

Proposition 2 (modularity of derivation) Let Π be a program. Let Pop be one of $\mathit{exit}, \mathit{fail}$. If

$$\mathit{call}\ G\,\langle\frac{\mathit{nil}}{\mathit{nil}}\rangle \rightarrowtriangle E_1 \rightarrowtriangle \ldots \rightarrowtriangle E_n \rightarrowtriangle \mathit{Pop}\ G\,\langle\frac{\mathbb{A}}{\mathit{nil}}\rangle$$

is a legal Π-derivation, then for every A-stack $\mathbb{U}$ and for every B-stack Σ such that *call* $G\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ is a legal event, holds:

$$\mathit{call}\ G\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle \rightharpoonup E_1 \ddagger \langle\tfrac{\Sigma}{\mathbb{U}}\rangle \rightharpoonup \ldots \rightharpoonup E_n \ddagger \langle\tfrac{\Sigma}{\mathbb{U}}\rangle \rightharpoonup \mathit{Pop}\ G\,\langle\tfrac{\Delta + \Sigma}{\mathbb{U}}\rangle$$

is also a legal Π-derivation. ■

Proof: Observe that our rules (with the exception of (S:conj:2)) refer only to the existence of the top element of some stack, never to the emptiness of a stack. Since the top element of a stack S cannot change after appending another stack to S, it is possible to emulate each of the original derivation steps using the 'new' stacks.

It remains to consider the rule (S:conj:2), which applies the whole current substitution upon the second conjunct. First note that any variables in a legal derivation stem either from the top-level goal or are fresh. According to the Lemma 4, a call event is always up-to-date, i. e. the current substitution has already been applied to the goal. The most general unifiers may be chosen to be idempotent, so a multiple application of a substitution amounts to a single application. Hence, if *call* $G\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ is a legal event, the substitution of Σ cannot affect any variables of the original derivation. □

5 Applications

5.1 Specifying program properties

Uniqueness and modularity of legal port derivations allow us to succinctly define some traditional notions.

Definition 11 (termination, success, failure) A goal G is said to terminate wrt program Π, if there is a Π-derivation

$$\mathit{call}\ G\,\langle\tfrac{nil}{nil}\rangle \overset{*}{\rightharpoonup} \mathit{Pop}\ G\,\langle\tfrac{\Delta}{nil}\rangle$$

where *Pop* is one of *exit*, *fail*. In case of *exit*, the derivation is *successful*, otherwise it is *failed*. ■

In a failed derivation, $\Delta = nil$.

Definition 12 (computed answer) In a successful derivation

$$\mathit{call}\ G\,\langle\tfrac{nil}{nil}\rangle \overset{*}{\rightharpoonup} \mathit{exit}\ G\,\langle\tfrac{\Delta}{nil}\rangle$$

is substOf(Δ), restricted upon the variables of G, called the *computed answer substitution* for G. ■

5.2 Proving program properties

Uniqueness of legal derivation steps enables *forward and backward* derivation steps, in the spirit of the Byrd's article. Push events (call, redo) are more amenable to forward steps, and pop events (exit, fail) are more amenable to backward steps. We illustrate this by a small example.

Lemma 5 If the events on the left-hand sides are legal, the following are legal derivations (for appropriate Σ_0, Σ_1):

$$\mathit{exit}\ A; B, \mathsf{fail}\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle \leftharpoondown \mathit{exit}\ A\,\langle\tfrac{\Sigma_0}{1/A;B,\mathsf{fail}\,\bullet\,\mathbb{U}}\rangle \tag{2}$$

$$\mathit{redo}\ A; B, \mathsf{fail}\,\langle\tfrac{\Sigma}{\mathbb{U}}\rangle \rightharpoonup \mathit{redo}\ A\,\langle\tfrac{\Sigma_1}{1/A;B,\mathsf{fail}\,\bullet\,\mathbb{U}}\rangle \tag{3}$$

■

Proof: The first statement claims: If *exit* $A; B, \mathsf{fail}\,\langle\frac{\Sigma}{\mathbb{U}}\rangle$ is legal, then it was reached via *exit* A. Without inspecting Σ, in general it is not known whether a disjunction succeeded via its first, or via its second member. But in this particular disjunction, the second member cannot succeed: Assume there are some $\mathbb{U}_0$, Σ_0 with *exit* $A; B, \mathsf{fail}\,\langle\frac{\Sigma}{\mathbb{U}}\rangle \leftharpoondown$ *exit* $B, \mathsf{fail}\,\langle\frac{\Sigma_0}{\mathbb{U}_0}\rangle$. According to the rules:

$$\mathit{exit}\ B, \mathsf{fail}\,\langle\tfrac{\Sigma_0}{\mathbb{U}_0}\rangle \leftharpoondown \mathit{exit}\ \mathsf{fail}\,\langle\tfrac{\Sigma_0}{2/B, \mathsf{fail} \bullet \mathbb{U}_0}\rangle \leftharpoondown \bot$$

So according to Lemma 3, *exit* $B, \mathsf{fail}\,\langle\frac{\Sigma_0}{\mathbb{U}_0}\rangle$ is not a legal event, which proves (2). Similarly, the non-legal derivation *redo* $B, \mathsf{fail} \rightharpoonup$ *redo* $\mathsf{fail} \rightharpoonup \bot$ proves (3). □

Modularity of legal derivations enables *abstracting the execution* of a goal, like in the following example.

Example 4 (modularity) Assume that a goal A succeeds, i. e. *call* $A\,\langle\frac{nil}{nil}\rangle \overset{*}{\rightharpoonup}$ *exit* $A\,\langle\frac{\Delta}{nil}\rangle$. Then we have the following legal derivation:

$$\begin{aligned}
\mathit{call}\ A, B\,\langle\tfrac{nil}{nil}\rangle &\rightharpoonup \mathit{call}\ A\,\langle\tfrac{nil}{1/A,B \bullet nil}\rangle, \text{ by (S:conj:1)}\\
&\overset{*}{\rightharpoonup} \mathit{exit}\ A\,\langle\tfrac{\Delta \bullet nil}{1/A,B \bullet nil}\rangle, \text{ by modularity and success of } A\\
&\rightharpoonup \mathit{call}\ B'\,\langle\tfrac{\Delta \bullet nil}{2/A,B \bullet nil}\rangle, \text{ by (S:conj:2), where } B' = \mathrm{substOf}(\Delta)(B)
\end{aligned}$$

If A fails, then we have:

$$\begin{aligned}
\mathit{call}\ A, B\,\langle\tfrac{nil}{nil}\rangle &\rightharpoonup \mathit{call}\ A\,\langle\tfrac{nil}{1/A,B \bullet nil}\rangle, \text{ by (S:conj:1)}\\
&\overset{*}{\rightharpoonup} \mathit{fail}\ A\,\langle\tfrac{nil}{1/A,B \bullet nil}\rangle, \text{ by modularity and failure of } A\\
&\rightharpoonup \mathit{fail}\ A, B\,\langle\tfrac{nil}{nil}\rangle, \text{ by (S:conj:3)}
\end{aligned}$$

❏

6 Conclusions and outlook

In this paper we give a simple mathematical definition S:PP of the 4-port model of pure Prolog. Some potential for formal verification of pure Prolog has been outlined. There are two interesting directions for future work in this area:

(1) formal specification of the control flow of *full Standard Prolog* (currently we have a prototype for this, within the 4-port model)

(2) formal specification and proof of some non-trivial program properties, like adequacy and non-interference of a practical program transformation.

7 Related work

Concerning attempts to formally define the 4-port model, we are aware of only few previous works. One is a graph-based model of Tobermann and Beckstein [TB93], who formalize the graph traversal idea of Byrd, defining the notion of a *trace* (of a given query with respect to a given program), as a path in a trace graph. The ports are quite lucidly defined as hierarchical nodes of such a graph. However, even for a simple recursive program and a ground query, with a finite SLD-tree, the corresponding trace graph is infinite, which limits its applicability. Another model of Byrd box is a continuation-based approach of Jahier, Ducassé and Ridoux [JDR00]. There is also a stack-based attempt in [Kul00], but although it provides for some parametrizing, it suffers essentially the same problem as the continuation-based approach, and also the prototypical implementation of the tracer given in [Byr80], taken as a specification of Prolog execution: In these three attempts, a port is represented by some semantic action (e. g. writing of a message), instead of a formal method. Therefore it is not clear how to use any of these models to prove some port-related assertions.

In contrast to the few specifications of the Byrd box, there are many more general models of pure (or even full) Prolog execution. Due to space limitations we mention here only some models, directly relevant to S:PP, and for a more comprehensive discussion see e. g. [KB01]. Comparable to our work are the stack-based approaches. Stärk gives in [Stä98], as a side issue, a simple operational semantics of pure logic programming. A state of execution is a stack of frame stacks, where each frame consists of a goal (ancestor) and an environment. In comparison, our state of execution consists of exactly one environment and one ancestor stack. The seminal paper of Jones and Mycroft [JM84] was the first to present a stack-based model of execution, applicable to pure Prolog with cut added. It uses a sequence of frames. In these stack-based approaches (including our previous attempt [KB01]), there is no *modularity*, i. e. it is not possible to abstract the execution of a subgoal.

Acknowledgments

Many thanks for helpful comments are due to anonymous referees.

References

[Byr80] Lawrence Byrd. Understanding the control flow of Prolog programs. In S. A. Tärnlund, editor, *Proc. of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, 1980. Also as D. A. I. Research Paper No. 151.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual)*. Springer-Verlag, 1996.

[JDR00] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Byrd's box model with a continuation semantics. In *Proc. of the WLPE'99, Las Cruces, NM*, volume 30 of *ENTCS*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume30.html.

[JM84] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proc. of the 1st Int. Symposium on Logic Programming (SLP'84)*, pages 281–288, Atlantic City, 1984.

[KB01] M. Kulaš and C. Beierle. Defining Standard Prolog in rewriting logic. In K. Futatsugi, editor, *Proc. of the 3rd Int. Workshop on Rewriting Logic and its Applications (WRLA 2000), Kanazawa*, volume 36 of *ENTCS*. Elsevier, 2001. http://www.elsevier.nl/locate/entcs/volume36.html.

[KL02] A. King and L. Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4):517–547, 2002.

[Kul00] M. Kulaš. A rewriting Prolog semantics. In M. Leuschel, A. Podelski, R. Ramakrishnan C. and U. Ultes-Nitsche, editors, *Proc. of the CL 2000 Workshop on Verification and Computational Logic (VCL 2000), London*, 2000.

[Lin95] T. Lindgren. Control flow analysis of Prolog (extended remix). Technical Report 112, Uppsala University, 1995. http://www.csd.uu.se/papers/reports.html.

[LO87] T. Lindholm and R. A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proc. of the 4th Int. Conference on Logic Programming (ICLP'87)*, pages 21–39, Melbourne, 1987.

[Stä98] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/~staerk/lptp.html.

[TB93] G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In *Proc. of the 1st Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'93), Linköping*, volume 749 of *LNCS*. Springer-Verlag, 1993.

A Leaving events

Leaving a call event

$$call\ A, B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ call\ A\ \langle \tfrac{\Sigma}{1/A,B\,\bullet\,\mathbb{U}} \rangle \tag{S:conj:1}$$

$$call\ A; B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ call\ A\ \langle \tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}} \rangle \tag{S:disj:1}$$

$$call\ \mathsf{true}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ exit\ \mathsf{true}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:true:1}$$

$$call\ \mathsf{fail}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ fail\ \mathsf{fail}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:fail}$$

$$call\ T_1 = T_2\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ \begin{cases} exit\ T_1 = T_2\ \langle \tfrac{\sigma\,\bullet\,\Sigma}{\mathbb{U}} \rangle, & \text{if } \mathrm{mgu}(T_1, T_2) = \sigma \\ fail\ T_1 = T_2\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle, & \text{otherwise} \end{cases} \tag{S:unif:1}$$

$$call\ G_A\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ \begin{cases} call\ \sigma(B)\ \langle \tfrac{\Sigma}{G_A\,\bullet\,\mathbb{U}} \rangle, & \text{if } H \text{ :- } B \text{ is a fresh renaming of a} \\ \quad \text{clause in } \Pi, \text{ and } \mathrm{mgu}(G_A, H) = \sigma, \text{ and } \sigma(G_A) = G_A \\ fail\ G_A\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle, & \text{otherwise} \end{cases} \tag{S:atom:1}$$

Leaving a redo event

$$redo\ A, B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ redo\ B\ \langle \tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}} \rangle \tag{S:conj:6}$$

$$redo\ A; B\ \langle \tfrac{OR(C,(N/A;B))\,\bullet\,\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ redo\ C\ \langle \tfrac{\Sigma}{N/A;B\,\bullet\,\mathbb{U}} \rangle \tag{S:disj:6}$$

$$redo\ \mathsf{true}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ fail\ \mathsf{true}\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:true:2}$$

$$redo\ T_1 = T_2\ \langle \tfrac{\sigma\,\bullet\,\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ fail\ T_1 = T_2\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:unif:2}$$

$$redo\ G_A\ \langle \tfrac{BY(B,G'_A)\,\bullet\,\Sigma}{\mathbb{U}} \rangle \ \rightharpoonup\ redo\ B\ \langle \tfrac{\Sigma}{G'_A\,\bullet\,\mathbb{U}} \rangle \tag{S:atom:4}$$

Leaving an exit event

$$exit\ A'\ \langle \tfrac{\Sigma}{1/A,B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ call\ B''\ \langle \tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}} \rangle, \text{ with } B'' := \mathrm{substOf}(\Sigma)(B) \tag{S:conj:2}$$

$$exit\ B'\ \langle \tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ exit\ A, B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:conj:4}$$

$$exit\ A\ \langle \tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ exit\ A; B\ \langle \tfrac{OR(A,(1/A;B))\,\bullet\,\Sigma}{\mathbb{U}} \rangle \tag{S:disj:4}$$

$$exit\ B\ \langle \tfrac{\Sigma}{2/A;B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ exit\ A; B\ \langle \tfrac{OR(B,(2/A;B))\,\bullet\,\Sigma}{\mathbb{U}} \rangle \tag{S:disj:5}$$

$$exit\ B\ \langle \tfrac{\Sigma}{G_A\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ exit\ G_A\ \langle \tfrac{BY(B,G_A)\,\bullet\,\Sigma}{\mathbb{U}} \rangle \tag{S:atom:2}$$

Leaving a fail event

$$fail\ A'\ \langle \tfrac{\Sigma}{1/A,B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ fail\ A, B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:conj:3}$$

$$fail\ B'\ \langle \tfrac{\Sigma}{2/A,B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ redo\ A\ \langle \tfrac{\Sigma}{1/A,B\,\bullet\,\mathbb{U}} \rangle \tag{S:conj:5}$$

$$fail\ A\ \langle \tfrac{\Sigma}{1/A;B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ call\ B\ \langle \tfrac{\Sigma}{2/A;B\,\bullet\,\mathbb{U}} \rangle \tag{S:disj:2}$$

$$fail\ B\ \langle \tfrac{\Sigma}{2/A;B\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ fail\ A; B\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:disj:3}$$

$$fail\ B\ \langle \tfrac{\Sigma}{G_A\,\bullet\,\mathbb{U}} \rangle \ \rightharpoonup\ fail\ G_A\ \langle \tfrac{\Sigma}{\mathbb{U}} \rangle \tag{S:atom:3}$$

B An example with variables

Assume the following program Π:

```
post(X,Y) :- one(X,Y), two(X,Y).
one(X,_) :- X=1.
two(_,Y) :- Y=a; Y=b.
```

Table 2 below shows the complete Π-execution of the goal post(X, Y), fail in the model S:PP. Highlighted are the A-stacks and the mgus. Notice the "lazy" binding of variables in the current goal.

call (post(X, Y), fail) , {*nil*} , {*nil*}

→ *call* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*nil*}

→ *call* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *call* one(X, Y) , {1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *call* X=1 , {one(X, Y) • 1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *exit* X=1 , {one(X, Y) • 1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {$[X/1]$ • *nil*}

→ *exit* one(X, Y) , {1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *call* two(1, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *call* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *call* Y=a , {(1/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *exit* Y=a , {(1/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {$[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *exit* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *exit* two(1, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *exit* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *exit* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *call* fail , {2/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *fail* fail , {2/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *redo* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *redo* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *redo* two(X, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=a, (1/(Y=a); Y=b)) • $[Y/a]$ • *BY*(X=1, one(X, Y)) • $[X/1]$ • *nil*}

→ *redo* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*OR*(Y=a, (1/(Y=a); Y=b)) • [Y/a] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* Y=a , {(1/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {[Y/a] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *fail* Y=a , {(1/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *call* Y=b , {(2/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *exit* Y=b , {(2/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {[Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *exit* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *exit* two(1, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *exit* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *exit* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *call* fail , {2/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *fail* fail , {2/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*BY*((one(X, Y), two(X, Y)), post(X, Y)) • *BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* two(X, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*((Y=a; Y=b), two(1, Y)) • *OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*OR*(Y=b, (2/(Y=a); Y=b)) • [Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* Y=b , {(2/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {[Y/b] • *BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *fail* Y=b , {(2/(Y=a); Y=b) • two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *fail* (Y=a; Y=b) , {two(1, Y) • 2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *fail* two(1, Y) , {2/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* one(X, Y) , {1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*BY*(X=1, one(X, Y)) • [X/1] • *nil*}

→ *redo* X=1 , {one(X, Y) • 1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {[X/1] • *nil*}

→ *fail* X=1 , {one(X, Y) • 1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *fail* one(X, Y) , {1/one(X, Y), two(X, Y) • post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *fail* (one(X, Y), two(X, Y)) , {post(X, Y) • 1/post(X, Y), fail • *nil*} , {*nil*}

→ *fail* post(X, Y) , {1/post(X, Y), fail • *nil*} , {*nil*}

→ *fail* (post(X, Y), fail) , {*nil*} , {*nil*}

Table 2. Execution of a query in S:PP