Electronic Notes in Theoretical Computer Science 36 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume36.html 17 pages

Defining Standard Prolog in Rewriting Logic

M. Kulaš and C. Beierle

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

Abstract

The coincidence between the model-theoretic and the procedural semantics of SLD-resolution does not carry over to a Prolog system that also implements non-logical features like cut and whose depth-first search strategy is incomplete. The purpose of this paper is to present the key concepts of a new, simple operational semantics of Standard Prolog in the form of rewriting rules. We use a novel linear representation of the Prolog tree traversal. A derivation is represented at the level of unification and backtracking. The rewriting system presented here can easily be implemented in a rewriting logic language, giving an executable specification of Prolog.

1 Introduction

There are numerous approaches dealing with Prolog's logical core, Horn clause programming, or various extensions of it. [2] is a classical paper dealing with SLD-resolution and defining an SLD-tree representation suitable for pure logic programming. However, the coincidence between the model-theoretic and the procedural semantics based on SLD-resolution does not carry over to a Prolog system that also implements extra-logical features and whose search strategy is typically incomplete. Many authors therefore abandon the classical, logical semantics and propose other semantics. The usual depth-first search employed in actual Prolog systems is modelled by the denotational semantics approaches of Jones and Mycroft [12], Nicholson and Foo [18], Debray and Mishra [8], de Vink [7] or Baudinet [4]. Among the first to specify backtracking and cut as an elegant use of continuations was also D. Schmidt [21]. The language constructs covered in these approaches include the cut operator, but further advanced control constructs like catch/throw or database predicates are not treated.

Continuations play a central role in denotational semantics, but it can also be beneficial to have them at the source level [6], [23]. Brisset and Ridoux [6] extend the semantics of [18] to cover new built-ins (for λProlog) that explicitly manipulate success and failure continuations. In [15] continuation passing is used for an intermediate representation for compilation to an ab-

stract machine. Our approach is also based on continuations, i. e. on a simple operational rendering of success/failure continuations.

Operational semantics of Horn clauses is studied in detail by Andrews in [1]. The semantics is expressed by rewriting systems that operate on backtracking stacks, i. e. sequences of pairs of substitutions (computed so far) and goals (still to be resolved). The work in [1] focusses on various combinations of parallel and sequential 'and' resp. 'or' search strategies for Horn logic; there is no treatment of extra-logical features like cut, negation-as-failure, etc. Operational semantics of Horn clauses has been given by [13] as well, within the implementation of constraint logic programming in ELAN. Backtracking is in [1],[13] modelled by nondeterministic choice.

The approaches cited above do not cover all of Prolog's built-ins. Two further approaches, [10] and [5], explicitly aim at covering full Standard Prolog. Deransart and Ferrand [10] introduce a logic programming based specification language together with an abstract data structure, the 'visited search tree', used for representing the semantics of a Prolog program. Based on these notions, the semantics of Prolog predicates is defined basically in terms of transitions of search trees. Whereas this approach uses logic programming as a full, but rather complex and somewhat low-level specification language, Börger and Rosenzweig [5] use abstract state machines (ASM), previously known as evolving algebras, to "provide a primary mathematical definition of Prolog". ASMs allow for high-level specifications, but the specification given in [5] is not easily executable by a standard tool like term rewriting.

The aim of the work reported here is to provide an intuitive, lean, easily executable semantics of full Prolog, including all its control constructs like meta-call, cut, if-then, catch, throw, all database operations like assert and retract. We came up with a simple, continuation-based operational model. Some features of our approach:

- a novel linear representation of the Prolog tree traversal, enabling comparatively small and readable specifications
- a deterministic representation of backtracking
- easy finding of ancestors, therefore suitable for representing cut and catch/throw in essentially the same, programmer-friendly way

The semantics is presented in the form of (conditional) rewriting rules of rewriting logic [17]. Currently we are implementing our rewriting system in Maude [1] . This paper presents the basic concepts of our approach (Sec. 2) and illustrates it on some of the key Prolog constructs, including extra-logical features like cut, catch and throw, as well as database update predicates (Sec. 3). In Sec. 4 we give two detailed examples. Section 5 rounds off the survey of related work. Section 6 points out some possible continuations of this work.

[1] we already have an implementation in Prolog

2 Representing derivation states and answers

A top-level goal G_0 we represent as

$$\mathbf{D}[\![G_0]\!] \bullet$$

with an operational reading *start the derivation of* G_0. The $\mathbf{D}$ is the *derivation operator*, $\mathbf{D}$-operator for short. Its argument in brackets represents the *current goal*. The fat point is marking the end of the whole derivation state.

2.1 Getting the first answer

The answer of a conforming Prolog processor is represented by non-reducible terms of our rewriting system. (We are talking about *the* answer here because of course a Prolog processor *is* deterministic.) Let us consider in turn the answer in case of success and in case of failure.

A successful derivation of a goal G_0 ends in a sequence

$$\boldsymbol{\sigma_1\sigma_2}\ldots\boldsymbol{\sigma_n}\,\mathbf{C}_{\Phi_{n-1}}[\![G_{n-1}]\!]\ldots\,\mathbf{C}_{\Phi_1}[\![G_1]\!]\,\mathbf{C}_{\Phi_0}[\![G_0]\!]\bullet$$

Here we have n substitutions at the beginning of the sequence, representing the most general unifiers (mgu's) used in the derivation. For each of the substitutions there is a *continuation*, depicted by the $\mathbf{C}$-operators, memorizing the still possible alternatives for its argument. Observe that the last continuation contains the original goal, the previous one contains the *resolvent* after selecting the first subgoal, and so on. The $\mathbf{C}$-operators may have an index, here depicted as Φ_i, $i = 0, ..., n-1$, which will be explained later (Sec. 3.1).

Thus, G_i is obtained from G_{i-1} by resolving the first subgoal with an appropriately renamed program clause, for each $i = 1, ..., n$, giving $\boldsymbol{\sigma_i}$. $G_n = \wedge$ is the empty word. Please note that we use postfix notation for *applying a substitution*, i. e. $X\boldsymbol{\sigma}$ means that the substitution $\boldsymbol{\sigma}$ is to be applied on the term X.

Such a successful derivation presents the computed answer substitution $\boldsymbol{\sigma_1}\circ\boldsymbol{\sigma_2}\circ\ldots\circ\boldsymbol{\sigma_n}$ (composition is from left to right). However, in our approach we want to show not only this final substitution, but all the unification and backtracking steps made during the derivation. Backtracking means reconsidering (cancelling) the already made choices (substitutions). Backtracking shall be depicted by the *undo-operator*, $\boldsymbol{u}$. The purpose of $\boldsymbol{u}$ is to cancel the immediately preceding substitution. Thus, in general, a derivation state as depicted above may contain, in front of any of the substitutions $\boldsymbol{\sigma_i}$, a *subsequence of cancelled substitutions* which looks like this (here $\boldsymbol{\rho_j}$ are substitutions):

$$\boldsymbol{\rho_1\rho_2}\ldots\boldsymbol{\rho_k}\underbrace{\boldsymbol{uu}\ldots\boldsymbol{u}}_{k}$$

In case of a successful derivation, the final derivation state will show a number of continuations equal to the number of non-cancelled substitutions.

For a failed derivation the picture is simple:

$$\boldsymbol{\sigma_1} \ldots \boldsymbol{\sigma_n} \underbrace{\boldsymbol{u} \ldots \boldsymbol{u}}_{k} \boldsymbol{u} \bullet$$

where k is the number of remaining substitutions in the sequence $\boldsymbol{\sigma_1} \ldots \boldsymbol{\sigma_n}$ that have not already been cancelled by a matching $\boldsymbol{u}$ to the left of $\boldsymbol{\sigma_n}$.

In the Sec. 3 we will develop a system *SP* for rewriting derivation states as an operational semantics for Prolog. If *SP* terminates on a derivation state $\mathcal{S}$, then $SP(\mathcal{S})$ denotes the unique normal form of $\mathcal{S}$ with respect to *SP*. For any derivation state $\mathcal{S}$ in normal form (with respect to *SP*) we define

$$\mathit{ExtractAnswer}(\mathcal{S})$$

by the exhaustive application of the following two rules[2] to $\mathcal{S}$.

$$\begin{aligned} \boldsymbol{\sigma}\ \boldsymbol{u} \quad &\Rightarrow \wedge \\ \mathbf{C}_{(\Phi)}[\![X]\!] \ &\Rightarrow \wedge \end{aligned}$$

where a rule having (Φ) as the index of a $\mathbf{D}$ or a $\mathbf{C}$ operator is actually an abbreviation for two rules, one where the operator does not have any index, and the other where the index is Φ, consistently throughout the rule.

The cumulative effect of the rules for $\mathit{ExtractAnswer}(\mathcal{S})$ is to erase all continuations (with or without indices) and all cancelled substitutions.

2.2 Moving on

In order to get not only the first answer substitution, we model a user's request for further answers. For any derivation state $\mathcal{S}$ in normal form (with respect to *SP*) we define

$$\mathit{StartBacktracking}(\mathcal{S})$$

by applying the rewrite rule

$$\boldsymbol{\sigma}\ \mathbf{C}_{(\Phi)}[\![X]\!] \Rightarrow \boldsymbol{\sigma}\ \boldsymbol{u}\ \mathbf{C}_{(\Phi)}[\![X]\!]$$

to $\mathcal{S}$. Inserting the undo-operator $\boldsymbol{u}$ initiates backtracking and corresponds exactly to the typing of ';' to a Prolog processor's top-level loop. Now we can again apply *SP* to $\mathit{StartBacktracking}(\mathcal{S})$. Therefore, we define the following:

If *SP* does not terminate on $\mathbf{D}[\![G]\!]\bullet$, then $\mathit{answer}_1(G)$ is undefined. Otherwise, we set

$$\begin{aligned} \mathit{answer}_1(G) &=_{\text{def}} \mathit{ExtractAnswer}(SP(\mathbf{D}[\![G]\!]\bullet)) \\ \mathit{next}_1(G) &=_{\text{def}} \mathit{StartBacktracking}(SP(\mathbf{D}[\![G]\!]\bullet)) \end{aligned}$$

[2] Recall that $\wedge$ denotes the empty word.

For $k > 1$, if $answer_{k-1}(G)$ is undefined or SP does not terminate on $next_{k-1}(G)$, then $answer_k(G)$ is undefined. Otherwise, we set

$$\begin{aligned} answer_k(G) &=_{\text{def}} ExtractAnswer(SP(next_{k-1}(G))) \\ next_k(G) &=_{\text{def}} StartBacktracking(SP(next_{k-1}(G))) \end{aligned}$$

So much to the initial and final states of a derivation. Actually, our objective is somewhat more general, to represent any *intermediate states* of a Prolog derivation, at the level of unification and backtracking.

A typical intermediate state will contain the derivation operator (precisely once) and an arbitrary number of continuations (corresponding to the number of non-cancelled substitutions, or exceeding it by one), and it will be reducible by one of the rules of the following section.

3 Rewriting derivation states

In this section we introduce our rewriting system *SP*. Let us start by showing how a resolution of a user-defined predicate looks like in our approach.

3.1 User-defined predicates

In case the predicate of the goal X is not built-in, its definition def(X) with respect to the user's program Π (i. e. the clauses defining the predicate of X) will be fetched, and the *choice points for X will be restricted* to this definition. This is expressed in the first rule below. The *restriction* will be carried as parameter to the derivation operator, here shown as index.

Note how the first rule implements the *logical update view* of Lindholm and O'Keefe [16], saying that the definition of a predicate shall be fixed at the time of its call.

$$\begin{aligned} \mathbf{D}[\![X, Y]\!] &\Rightarrow \mathbf{D}_{\Phi}[\![X, Y]\!] \text{ if } \neg\text{builtin}(X) \text{ and } \Phi = \text{def}(X) \\ \mathbf{D}_{\Phi}[\![X, Y]\!] &\Rightarrow \boldsymbol{\sigma}\ \mathbf{D}[\![(R, Y)\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\Phi'}[\![X, Y]\!] \text{ if resolve}(X,\Phi) = <\boldsymbol{\sigma}\ R\ \Phi'> \\ \mathbf{D}_{\Phi}[\![X, Y]\!] &\Rightarrow \boldsymbol{u} \text{ if resolve}(X,\Phi) = \boldsymbol{u} \end{aligned}$$

The second rule implements the actual resolution step. If our goal can be resolved with respect to the restriction Φ, giving a resolvent R and the most general unifier $\boldsymbol{\sigma}$, then the current goal will be replaced in the expected way.

Furthermore, the derivation state will be enhanced by the substitution $\boldsymbol{\sigma}$ to the left of the **D**-operator, and by a *continuation* to the right of it. Namely, we must remember the possible further ways to resolve the goal, represented by the rest of the restriction Φ after removing the non-applicable clauses as well as the consumed ones so far. This leaves us with the choice points Φ'. In case of backtracking, the alternative derivation paths shall be activated, but

for the time being they are dormant, represented by the *continuation operator* **C** (with or without an index).

The third rule handles the case that our goal cannot be resolved. Then the current goal will be erased altogether and replaced by a special *undo-operator* $\boldsymbol{u}$ meaning that backtracking shall be started. Notice that the **D**-operator has vanished. So now the **C**-operators must take over. For an illustration, see Sec. 4.1.

3.2 Backtracking

The following three rules implement backtracking. The first one handles the case that the continuation wasn't restricted (there is no index to the **C**-operator).

$$\begin{aligned}
\boldsymbol{u}\,\mathbf{C}[\![X]\!] &\Rightarrow \boldsymbol{u}\,\mathbf{D}[\![X]\!] \\
\boldsymbol{u}\,\mathbf{C}_{\Phi}[\![X]\!] &\Rightarrow \boldsymbol{u}\,\mathbf{D}_{\Phi}[\![X]\!] \quad \text{if } \Phi \neq \emptyset \\
\boldsymbol{u}\,\mathbf{C}_{\emptyset}[\![X]\!] &\Rightarrow \boldsymbol{u}\,\boldsymbol{u}
\end{aligned}$$

In case there was a restriction, it has to be propagated. This is the meaning of the second rule. The third rule handles the special case that a continuation is *empty*, i. e. there are no choice points left. This is represented by the empty set $\emptyset$ as the index of the continuation.

What do these rules mean? Quite simply, if there is a non-empty continuation immediately to the right of a $\boldsymbol{u}$, then it shall be *activated*, i. e. turned into the current goal. [3]

3.3 Some easy logic-and-control, part I: true, fail, halt

The first two rules are straightforward. The last two are a bit more interesting: due to the fact that the meta-operators Halt and Error are nowhere further specified, by any other rules, and the **D**-operator is gone, the net effect of each of these two rules is to stop the computation immediately, as intended.

$$\begin{aligned}
\mathbf{D}[\![\mathit{true}, X]\!] &\Rightarrow \mathbf{D}[\![X]\!] \\
\mathbf{D}[\![\mathit{fail}, X]\!] &\Rightarrow \boldsymbol{u} \\
\mathbf{D}[\![\mathit{halt}, X]\!] &\Rightarrow \text{Halt} \\
\mathbf{D}[\![X, Y]\!] &\Rightarrow \text{Error if } \mathrm{var}(X)
\end{aligned}$$

[3] Actually we do not need to keep the $\boldsymbol{u}$'s in the derivation state at all. Instead we could have postulated the following rule $\boldsymbol{\sigma u}\mathbf{C}[\![X]\!] \Rightarrow \mathbf{D}[\![X]\!]$, as well as $\boldsymbol{\sigma u}\mathbf{C}_{\emptyset}[\![X]\!] \Rightarrow \boldsymbol{u}$, to cancel all the 'fallible' substitutions right away. But in this paper we are simply giving our most detailed view of a derivation, and not considering any optimizations.

Of course, error handling of the built-in predicates can be more refined, but here we disregard this issue.

Now at the latest we have to introduce a synactical convention we already used: Whenever we write a conjunctive goal X, Y as the argument of our operators, the tail of this conjunction may be empty, in which case the until now undisclosed $\wedge$-rule takes over:

$$\mathbf{D}[\![\wedge]\!] \;\Rightarrow\; \wedge$$

Here we slightly abuse the Prolog syntax for the sake of readability. In practice this is not a problem, since one would more likely represent a derivation state as a *list* than as a conjunction anyway.

3.4 Term unification

This should be straightforward by now.

$$\mathbf{D}[\![(X = Y), Z]\!] \;\Rightarrow\; \boldsymbol{\sigma}\, \mathbf{D}[\![Z\boldsymbol{\sigma}]\!]\, \mathbf{C}_{\emptyset}[\![(X = Y), Z]\!] \;\text{ if unify}(X,Y) = \boldsymbol{\sigma}$$

$$\mathbf{D}[\![(X = Y), Z]\!] \;\Rightarrow\; \boldsymbol{u} \;\text{ if unify}(X,Y) = \boldsymbol{u}$$

3.5 Meta-call, first draft

As a first try we might define the meta-call like this

$$\mathbf{D}[\![\mathit{call}(X), Y]\!] \;\Rightarrow\; \mathbf{D}[\![X, Y]\!]$$

but that would be wrong, strictly speaking. Meta-call would behave correctly, except that it would be *transparent for cut*, meaning a cut within the meta-call would go too far. See [19] for a comprehensive discussion of cut and the problem of transparency.

3.6 Meta-call, correct version (opaque for cut)

To rectify this, we use a dummy 'insulating layer', an empty continuation. This should be clear from the specification of cut in the following (Sec. 3.8).

$$\mathbf{D}[\![\mathit{call}(X), Y]\!] \;\Rightarrow\; \boldsymbol{\varepsilon}\, \mathbf{D}[\![X, Y]\!]\, \mathbf{C}_{\emptyset}[\![\mathit{call}(X), Y]\!]$$

The symbol $\boldsymbol{\varepsilon}$ denotes the trivial (identity) substitution. We need it in several rules in order to preserve the correspondence between the substitutions and the continuations resp. the undo-operators $\boldsymbol{u}$.

This is necessary because we need to know, in case of backtracking, which substitution precisely is going to be undone. So in case the rule introduces a choice point (represented by a continuation), there has to be a corresponding substitution, even a trivial one, $\boldsymbol{\varepsilon}$.

3.7 Some easy logic-and-control, part II: repeat, negation

Here we show some 'derived' built-ins, i. e. some that can be defined in terms of others. Strictly speaking, they do not need an own specification, since they can be treated like user-defined, but are presented here nevertheless as exercises.

$$\mathbf{D}[\![\mathit{repeat}, X]\!] \;\Rightarrow\; \varepsilon\, \mathbf{D}[\![X]\!]\, \mathbf{C}[\![\mathit{repeat}, X]\!]$$

$$\mathbf{D}[\![\backslash{+}(X), Y]\!] \;\Rightarrow\; \varepsilon\, \mathbf{D}[\![\mathit{call}(X), !, \mathit{fail}]\!]\, \mathbf{C}[\![Y]\!]$$

3.8 Cut

The idea is quite intuitive, namely the parent of the cut sequence must be found and all the choices underway made deterministic, i. e. all the continuations emptied. That is all. To find the parent, we take advantage of our Prolog tree representation, allowing for a very simple *syntactic criterion*, namely:

Suffix criterion *Let the current goal be X, Y. To find the parent of X, look for the leftmost continuation whose argument C does not end in X, Y (more formally: X, Y is not an instance of a proper suffix of C).*

As to the actual specification, a few more words are necessary, due to the rewriting formalism. We found specifying in Prolog here of advantage, having subroutines that work upon the *whole* input sequence, instead of a definite portion. But the subroutines we need here are not difficult to simulate in rewriting, due to the fact that they are very simple. When the first literal in the current goal is a cut, we basically have to start a sub-routine 'cutting out' all the alternatives (represented here by emptying the continuations) until the parent of the current goal has been found. On finding the parent, its continuation shall also be emptied, and that concludes the sub-routine, so we can continue deriving the rest of the current goal. We implemented this in rewriting by means of several *markers*, namely:

Pending(X)	$\rightsquigarrow$ place-holder
Pending(X) Parent($(!, Y)$)	$\rightsquigarrow$ start the subroutine for cut and mark the entry point
Parent(X)	$\rightsquigarrow$ search the parent of X, removing choices underway
Return	$\rightsquigarrow$ now that you have found the parent, go back

Here are the actual rewriting rules[4] for cut.

[4] Recall that a rule having (Φ) as the index of a continuation is actually an abbreviation for two rules, one where the continuation does not have any index, and the other where the index is Φ.

$$
\begin{array}{lll}
\mathbf{D}[\![!, X]\!] & \Rightarrow & \text{Pending}(\mathbf{D}[\![X]\!]\,)\ \text{Parent}((!, X)) \\
\text{Parent}(G)\ \mathbf{C}_{(\Phi)}[\![Has]\!] & \Rightarrow & \mathbf{C}_{\emptyset}[\![Has]\!]\ \text{Parent}(G)\ \ \text{if suffix}(Has,G) \\
\text{Parent}(G)\ \mathbf{C}_{(\Phi)}[\![Hasnt]\!] & \Rightarrow & \text{Return}\ \mathbf{C}_{\emptyset}[\![Hasnt]\!]\ \ \text{if } \neg\text{suffix}(Hasnt,G) \\
\text{Parent}(G)\ \bullet & \Rightarrow & \text{Return} \\
\mathbf{C}_{(\Phi)}[\![X]\!]\ \text{Return} & \Rightarrow & \text{Return}\ \mathbf{C}_{(\Phi)}[\![X]\!] \\
\text{Pending}(G)\ \text{Return} & \Rightarrow & G
\end{array}
$$

Observe that we do not need a special 'cut-marker' [12,8,4,7] or 'cut-parent' [5], because in our approach the cut-parent is literally *the parent of the cut (starting the current goal)*, and the parent of *any* goal can in our approach be found by the simple syntactic criterion from above, the suffix-criterion. Incidentally, note that, due to the intricacies of the standard specification of if-then-else, as in [9], we had to be careful with our specification of it, see [14], so that the cut-parent may indeed uniformly be the parent of the current cut.

A collateral advantage of our representation of the Prolog tree is that the algorithm for cut can be easily generalized for throw.

3.9 Catch and throw

To find the right node in the Prolog tree to go to upon encountering $throw(Ball)$, we have to find, according to [9], "the closest ancestor node whose chosen predication has the form $catch(Goal, Catcher, Recovergoal)$, which is still executing its *Goal* argument and such that a freshly renamed copy $Ball'$ of *Ball* unifies with *Catcher*".

What this amounts to in our representation, is: search along the chain of parents (by the suffix-criterion, Sec. 3.8) for the goal starting with $catch(G, B, R)$.

$$
\begin{array}{lll}
\mathbf{D}[\![catch(G,B,R), X]\!] & \Rightarrow & \boldsymbol{\varepsilon}\ \mathbf{D}[\![G, X]\!]\ \mathbf{C}[\![catch(G,B,R), X]\!] \\
\mathbf{D}[\![throw(B), X]\!] & \Rightarrow & \text{ThrowAnc}((throw(B), X),(throw(B), X)) \\
\text{ThrowAnc}(T,X)\ \mathbf{C}_{(\Phi)}[\![C]\!] & \Rightarrow & \boldsymbol{u}\ \text{ThrowAnc}(T,X)\ \ \text{if suffix}(C,X) \\
\text{ThrowAnc}((throw(B), Y), X)\ \mathbf{C}[\![C]\!] & \Rightarrow & \boldsymbol{u}\,\boldsymbol{\sigma}\mathbf{D}[\![(R, Z)\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\emptyset}[\![C]\!]\ \ \text{if } \neg\text{suffix}(C,X) \\
\multicolumn{3}{l}{\qquad \text{and } C = (catch(G, B', R), Z) \text{ and unify}(\text{fresh}(B),B') = \boldsymbol{\sigma}} \\
\text{ThrowAnc}(T,X)\ \mathbf{C}_{(\Phi)}[\![C]\!] & \Rightarrow & \boldsymbol{u}\ \text{ThrowAnc}(T,C)\ \ \text{if } \neg\text{suffix}(C,Y) \\
\multicolumn{3}{l}{\qquad \text{and } \big(C \neq (catch(G, B', R), Z) \text{ or } \Phi = \emptyset\ \text{ or unify}(\text{fresh}(B),B') = \boldsymbol{u}\big)} \\
\text{ThrowAnc}(T,X)\ \bullet & \Rightarrow & \text{Error} \\
\boldsymbol{u}\ \mathbf{C}[\![catch(G,B,R), Z]\!] & \Rightarrow & \boldsymbol{u}\,\boldsymbol{u}
\end{array}
$$

3.10 Clause creation and destruction

To specify the builtins responsible for database updates, we add the user's program Π to our derivation state, making it a *global parameter* of the derivation. So we would actually start with the following derivation state:

$$\mathbf{P}[\![\Pi]\!]\,\mathbf{D}[\![G_0]\!]\,\bullet$$

$$\begin{aligned}
\mathbf{D}[\![\mathit{asserta}(K), X]\!] &\;\Rightarrow\; \mathrm{Update}(\mathit{asserta}(K))\ \mathrm{Pending}(\mathbf{D}[\![\mathit{asserta}(K), X]\!]\,)\\
\mathbf{D}[\![\mathit{retract}(K), X]\!] &\;\Rightarrow\; \mathrm{Update}(\mathit{retract}(K))\ \mathrm{Pending}(\mathbf{D}[\![\mathit{retract}(K), X]\!]\,)
\end{aligned}$$

$$\begin{aligned}
\mathbf{P}[\![\Pi]\!]\ \mathrm{Update}(\mathit{asserta}(K)) &\;\Rightarrow\; \mathbf{P}[\![\Pi']\!]\ \mathrm{Return}(\boldsymbol{\varepsilon}) \ \text{ if addaclause}(K,\Pi) = \Pi'\\
\mathbf{P}[\![\Pi]\!]\ \mathrm{Update}(\mathit{retract}(K)) &\;\Rightarrow\; \mathbf{P}[\![\Pi']\!]\ \mathrm{Return}(\boldsymbol{\sigma}) \ \text{ if delfirstclause}(K,\Pi) = <\boldsymbol{\sigma}\,\Pi'>\\
\mathbf{P}[\![\Pi]\!]\ \mathrm{Update}(\mathit{retract}(K)) &\;\Rightarrow\; \mathbf{P}[\![\Pi]\!]\ \mathrm{Return}(\boldsymbol{u}) \ \text{ if delfirstclause}(K,\Pi) = \boldsymbol{u}\\
X\ \mathrm{Update}(\mathit{How}) &\;\Rightarrow\; \mathrm{Update}(\mathit{How})\,X \ \text{ if } X \neq \mathbf{P}[\![.]\!]
\end{aligned}$$

$$\begin{aligned}
\mathrm{Return}(Z)\ \mathrm{Pending}(\mathbf{D}[\![\mathit{asserta}(K), X]\!]\,) &\;\Rightarrow\; \mathbf{D}[\![X]\!]\\
\mathrm{Return}(\boldsymbol{\sigma})\ \mathrm{Pending}(\mathbf{D}[\![\mathit{retract}(K), X]\!]\,) &\;\Rightarrow\; \boldsymbol{\sigma}\,\mathbf{D}[\![X\boldsymbol{\sigma}]\!]\,\mathbf{C}[\![\mathit{retract}(K), X]\!]\\
\mathrm{Return}(\boldsymbol{u})\ \mathrm{Pending}(\mathbf{D}[\![\mathit{retract}(K), X]\!]\,) &\;\Rightarrow\; \boldsymbol{u}\\
\mathrm{Return}(Z)\ X &\;\Rightarrow\; X\ \mathrm{Return}(Z) \ \text{ if } X \neq \mathrm{Pending}(.)
\end{aligned}$$

Good news: All the rules introduced so far remain the same, and in fact the first rule of Sec. 3.1 relies upon some such program representation in order to fetch the definition $\mathrm{def}(X)$ of a user-defined goal X.

Here we recycled the place-holding marker Pending from Sec. 3.8, and introduced a further marker Update whose purpose is to find and update the program. The subroutine-exiting marker Return (cf. Sec. 3.8) is used to bring the resulting substitution back to the pending goal.

3.11 Disjunction, first draft

Until now we considered only 'atomary' goals, i. e. literals. But the Prolog syntax allows for embedded disjunction. The first try could be to specify disjunction as

$$\mathbf{D}[\![(X;Y), Z]\!] \;\Rightarrow\; \boldsymbol{\varepsilon}\,\mathbf{D}[\![X, Z]\!]\,\mathbf{C}[\![Y, Z]\!]$$

But that wouldn't be correct, because now the transparence for cut is gone. (If disjunction were opaque for cut, a typical repeat-cut-fail loop would never terminate.)

3.12 Disjunction, correct version (transparent for cut)

So how to specify disjunction so as to be transparent for cut, *and* at the same time preserve our simple suffix-criterion? Seems impossible, but there is a way to it. For this purpose we re-interpret the syntax of the goal (with respect to the Prolog syntax) so that the interpunction signs $'($, $';'$ and $')'$ acquire a new meaning: they can now be *goals*. Therefore, our original goal can now, syntactically, *only be a conjunction*.

More precisely, we perform a source-to-source transformation ('conjunctifying') which takes as input an arbitrary Prolog goal, and produces a flat conjunction of literals, enriched with the three *special goals* $'($, $';'$ and $')'$ used solely for disjunction. Thus, the disjunction $X;Y$ is transformed into the conjunction $'(', X, ';', Y, ')'$. Obviously, apart from adding or removing some brackets, this transformation does not change the sequence of symbols making up the goal ('the looks' of it), it just 'dissolves' the disjunction. Assuming a disjunction is given in the dissolved form, we specify its meaning by the following four rewriting rules:

$$\begin{aligned}
&\mathbf{D}[\![(, X, ;, Y,), \mathit{Rest}]\!] && \Rightarrow\ \boldsymbol{\varepsilon}\,\mathbf{D}[\![X, ;, Y,), \mathit{Rest}]\!]\,\mathbf{C}[\![; X, ;, Y,), \mathit{Rest}]\!]\\
&\mathbf{D}[\![; , Y,), \mathit{Rest}]\!] && \Rightarrow\ \mathbf{D}[\![), \mathit{Rest}]\!]\\
&\mathbf{D}[\![), \mathit{Rest}]\!] && \Rightarrow\ \mathbf{D}[\![\mathit{Rest}]\!]\\
&\boldsymbol{u}\,\mathbf{C}[\![; , X, ;, Y,), \mathit{Rest}]\!] && \Rightarrow\ \boldsymbol{u}\,\mathbf{D}[\![Y,), \mathit{Rest}]\!]
\end{aligned}$$

It should be obvious that these rules indeed specify disjunction correctly. Transparence for cut should also be obvious, because the continuation in the first rule has the current goal as its suffix. For an illustration see Sec. 4.2.

Good news: All the rules given in the previous sections *remain the same* when switching to the *dissolved form* of a goal, instead of the usual Prolog syntax. Only the first rule for if-then-else (see [14]) must be adapted for the enhanced syntax, namely the first rules of disjunction and if-then-else have to be blended like this:

$$\begin{aligned}
&\mathbf{D}[\![(, \mathit{If} \rightarrow \mathit{Then}, ;, \mathit{Else},), Z]\!] && \Rightarrow\ \mathbf{D}[\![(, \mathit{once}(\mathit{If}),\ \mathrm{ThisBranch}(\mathit{once}(\mathit{If}))\,, \mathit{Then}, ;, \mathit{Else},), Z]\!]\\
&\mathbf{D}[\![(, X, ;, Y,), Z]\!] && \Rightarrow\ \boldsymbol{\varepsilon}\,\mathbf{D}[\![X, ;, Y,), Z]\!]\,\mathbf{C}[\![; X, ;, Y,), Z]\!]\ \text{ if } X \neq (\mathit{If} \rightarrow \mathit{Then})
\end{aligned}$$

The first rule handles the special case of a disjunction which is also an if-then-else, eliminating the if-then. The second rule handles the general case.

4 Some examples

4.1 Horn clause programming

Let our program Π consist of the following three pure Prolog clauses K_1, K_2 and K_3.

p(1) :- p(2), p(3). % K_1

p(2) :- p(4). % K_2

p(4). % K_3

We show a step-by-step derivation of a non-deterministic goal $p(X)$.

$\mathbf{D}[\![p(X)]\!]$

$\Rightarrow \mathbf{D}_\Phi[\![p(X)]\!]$ by 3.1, rule 1, Φ={K_1, K_2, K_3}

$\Rightarrow \boldsymbol{\sigma_1}\mathbf{D}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 2, $\boldsymbol{\sigma_1}$={X/1}, Φ'={K_2, K_3}

$\Rightarrow \boldsymbol{\sigma_1}\mathbf{D}_\Phi[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 1

Note that we used the same index as in the second line, which is correct since def(p(X))=def(p(2))={K_1, K_2, K_3}.

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\mathbf{D}[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 2, Φ''={K_3}

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\mathbf{D}_\Phi[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 1

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\mathbf{D}[\![true, p(3)]\!]\,\mathbf{C}_\emptyset[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 2

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\mathbf{D}[\![p(3)]\!]\,\mathbf{C}_\emptyset[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.3, rule 1

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\mathbf{D}_\Phi[\![p(3)]\!]\,\mathbf{C}_\emptyset[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 1

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\mathbf{C}_\emptyset[\![p(4), p(3)]\!]\,\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 3

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\mathbf{C}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.2, rule 3

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\mathbf{D}_{\Phi''}[\![p(2), p(3)]\!]\,\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.2, rule 2

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\mathbf{C}_{\Phi'}[\![p(X)]\!]$ by 3.1, rule 3

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\mathbf{D}_{\Phi'}[\![p(X)]\!]$ by 3.2, rule 2

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\boldsymbol{\sigma_2}\mathbf{D}[\![p(4)]\!]\,\mathbf{C}_{\Phi''}[\![p(X)]\!]$ by 3.1, rule 2, $\boldsymbol{\sigma_2}$={X/2}

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\boldsymbol{\sigma_2}\mathbf{D}_\Phi[\![p(4)]\!]\,\mathbf{C}_{\Phi''}[\![p(X)]\!]$ by 3.1, rule 1

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\boldsymbol{\sigma_2}\boldsymbol{\varepsilon}\mathbf{D}[\![true]\!]\,\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\Phi''}[\![p(X)]\!]$ by 3.1, rule 2

$\Rightarrow \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\boldsymbol{\sigma_2}\boldsymbol{\varepsilon}\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\Phi''}[\![p(X)]\!]$ by 3.3, rule 1

No further rules of *SP* are applicable, so this is the normal form of our initial derivation state $\mathbf{D}[\![\text{p(X)}]\!]\bullet$ with respect to *SP*, or more formally:

$$SP(\mathbf{D}[\![p(X)]\!]\bullet) = \boldsymbol{\sigma_1}\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}\boldsymbol{u}\boldsymbol{u}\boldsymbol{u}\boldsymbol{\sigma_2}\boldsymbol{\varepsilon}\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\Phi''}[\![p(X)]\!]\bullet$$

(We left out the end-marker $\bullet$ from the above steps because it didn't play an active part in this particular derivation.)

According to our simulation of a conforming Prolog processor (Sec. 2), the first computed answer substitution shall be obtained from $answer_1(p(X))$

$$\begin{aligned} answer_1(p(X)) &= ExtractAnswer(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon} \mathbf{C}_\emptyset[\![p(4)]\!]\, \mathbf{C}_{\Phi''}[\![p(X)]\!]\ \bullet) \\ &= \boldsymbol{\sigma_2 \varepsilon}\ \bullet \end{aligned}$$

by composing the substitutions, finally giving $\boldsymbol{\sigma_2}$ as the computed answer substitution, which is obviously correct here.

What if we want further answers? Well, on the top-level of a Prolog processor we would type a semicolon ';'. In our model this corresponds to rewriting, with respect to *SP*, the sequence $next_1(p(X))$

$$\begin{aligned} next_1(p(X)) &= StartBacktracking(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon} \mathbf{C}_\emptyset[\![p(4)]\!]\, \mathbf{C}_{\Phi''}[\![p(X)]\!]\ \bullet) \\ &= \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon u} \mathbf{C}_\emptyset[\![p(4)]\!]\, \mathbf{C}_{\Phi''}[\![p(X)]\!]\ \bullet \end{aligned}$$

So the sequence generated by $next_1(p(X))$ is the new starting state for *SP*.

$\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon u} \mathbf{C}_\emptyset[\![p(4)]\!]\, \mathbf{C}_{\Phi''}[\![p(X)]\!]$

$\Rightarrow\ \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu} \mathbf{C}_{\Phi''}[\![p(X)]\!]$ by 3.2, rule 3

$\Rightarrow\ \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu} \mathbf{D}_{\Phi''}[\![p(X)]\!]$ by 3.2, rule 2

$\Rightarrow\ \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3} \mathbf{D}[\![true]\!]\, \mathbf{C}_{\Phi'''}[\![p(X)]\!]$ by 3.1, rule 2, $\boldsymbol{\sigma_3}$={X/4}, $\Phi'''=\emptyset$

$\Rightarrow\ \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3} \mathbf{C}_\emptyset[\![p(X)]\!]$ by 3.3, rule 1

The second answer will be given by

$$\begin{aligned} answer_2(p(X)) &= ExtractAnswer(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3} \mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet) \\ &= \boldsymbol{\sigma_3}\ \bullet \end{aligned}$$

The computation of further answers will start from the sequence

$$\begin{aligned} next_2(p(X)) &= StartBacktracking(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3} \mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet) \\ &= \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3 u} \mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet \end{aligned}$$

and won't last long:

$\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 uu \sigma_3 u} \mathbf{C}_\emptyset[\![p(X)]\!]$

$\Rightarrow\ \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 uu \sigma_3 uu}$ by 3.2, rule 3

giving as the third answer

$$\begin{aligned} answer_3(p(X)) &= ExtractAnswer(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 uu \sigma_3 uu}\ \bullet) \\ &= \boldsymbol{u}\ \bullet \end{aligned}$$

meaning that there is no further answer substitution, so the computation of $p(X)$ is finished.

4.2 Repeat, disjunction, cut

Let our program Π consist of the following three clauses.

q :– repeat, p(X), (X=b, !; fail).

p(a).

p(b).

We illustrate the vital importance of the disjunction's cut transparency for the universal termination of the repeat loop.

$$
\begin{aligned}
&\mathbf{D}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon}\mathbf{D}[\![\mathit{repeat}, p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon}\mathbf{D}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}, p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1}\mathbf{D}[\![\mathit{true}, (a = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}, p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1}\mathbf{D}[\![(a = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon}\mathbf{D}[\![a = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![; a = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon u}\mathbf{C}[\![; a = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon u}\mathbf{D}[\![\mathit{fail})]\!]\, \mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu}\mathbf{C}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu}\mathbf{D}_{\Phi}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu\sigma_2}\mathbf{D}[\![(b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu\sigma_2\varepsilon}\mathbf{D}[\![b = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![; b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![p(X), (X = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu\sigma_2\varepsilon\varepsilon}\mathbf{D}[\![!; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![b = b, !; \mathit{fail})]\!]\, \mathbf{C}[\![; b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![p(X)...]\!]\, \mathbf{C}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu\sigma_2\varepsilon\varepsilon}\mathbf{D}[\![; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![; b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![p(X)...]\!]\, \mathbf{C}_{\emptyset}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!] \\
&\Rightarrow \boldsymbol{\varepsilon\varepsilon\sigma_1\varepsilon uu\sigma_2\varepsilon\varepsilon}\mathbf{C}_{\emptyset}[\![b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![; b = b, !; \mathit{fail})]\!]\, \mathbf{C}_{\emptyset}[\![p(X)...]\!]\, \mathbf{C}_{\emptyset}[\![\mathit{repeat}...]\!]\, \mathbf{C}_{\emptyset}[\![q]\!]
\end{aligned}
$$

Notice, in the third and the second line from below, how the suffix criterion 'swipes clean' the continuations up to, and including, the one for *repeat*. This is essential for the termination of the loop.

5 Related approaches (coda)

To our knowledge, the first operational semantics for pure Prolog with cut was proposed by Jones and Mycroft in [12] (also used in [8]). The operational model of [12], a structure-sharing interpreter, employs, like we do, the idea

of memorizing alternative computation paths by means of 'filing' the current goal plus the rest of the definition for its leftmost literal. But, by simplifying some design decisions of [12], on the questions of what the current goal *is*, and how to handle the variables, we obtained a model which appears to be both simpler and more expressive. To accomodate cut, [12] had to enrich their model of pure Prolog (by an additional stack per subgoal). It was not clear how to accomodate e. g. catch/throw, or findall.

More specifically, in [12] the *current goal* does not mean the rest of the path, as here, but rather just the *current subgoal* (this can be: a top-level goal, or a body of a clause). The advantages of this representation begin to dwindle when cut is a part of the language. To accomodate cut, in [12] each subgoal has to bookkeep an additional 'dump' stack, which is a rest of the whole state. Dump stack is also used in [7]. As opposed to this, our concept of the current goal enables the simple suffix-criterion, giving the cut and catch/throw for free, i. e. without any additional bookkeeping. Also, for the purposes of renaming and computing answer substitutions, [12] needs for each path bookkeeping of the 'current substitution' and, in [8], for each subgoal the 'project variables' (those relevant for the parent goal). In our approach unifiers are not 'composed away' into the current substitution, so for computing the answer substitutions it suffices to bookkeep, once for the whole state, the sequence of unifiers: a forward step inserts a unifier, and a backward step cancels a unifier. This suffices also for renaming, which we didn't elaborate in this paper, but is implicit in resolve(X,Φ) and delfirstclause(K,Π), as well as explicit, in fresh(X), which is used for catch/throw and findall. So it is sufficient to have the substitutions *as a global parameter*, in the same manner as the user's program is treated, thereby further reducing bookkeeping.

6 Conclusions and further work

We presented a new, simple operational semantics for Standard Prolog. The first main novelty is a non-traditional representation of success/failure continuations, used for a liner rendering of the Prolog computation tree. In particular, it allows for the simple *suffix-criterion*, which enables the specification of cut or catch/throw without the need for cut-markers or other additional mechanisms. The second main novelty is the simple way of handling variables, due to not composing the unifiers. In addition to the Prolog features treated in this paper, in [14] we cover several more builtin predicates, rounding up on what is commonly seen as 'the' characteristic subset of Standard Prolog (logic and control, database update and all-solution predicates).

Since, in this paper, by *Prolog* we refer to the full language including all its extralogical features, we cannot expect to establish a complete correspondence between the well-known results about the model-theoretic, fixpoint and SLD-resolution based semantics of (pure) logic programming [2] and our operational semantics. However, when restricted to pure Prolog, our operational seman-

tics precisely reflects the SLD-resolution semantics based on SLD-trees with a depth-first, left-to-right search strategy. For example, assume a pure Prolog program and a (pure Prolog) goal G. If SP terminates on $\mathcal{S} = \mathbf{D}[\![G]\!] \bullet$ and $ExtractAnswer(SP(\mathcal{S})) = \boldsymbol{\sigma_1}\boldsymbol{\sigma_2}...\boldsymbol{\sigma_n} \bullet$, then the composition $\boldsymbol{\sigma} = \boldsymbol{\sigma_1} \circ \boldsymbol{\sigma_2} \circ ... \circ \boldsymbol{\sigma_n}$ is a correct answer substitution [2] for G. Moreover, the sequence of answer substitutions given by $answer_1(G)$, $answer_2(G)$, $answer_3(G), ...$ corresponds exactly to the sequence of solutions produced by a standard conforming Prolog processor [11].

A natural future objective is to construct a denotational semantics corresponding to the operational semantics given here, as exemplified by [12], [8] and [7]. This builds the basis for validation. Further we plan to investigate potentials of our approach in verification of program properties.

Acknowledgments

Many thanks forvaluable comments are due to the anonymous referees.

References

[1] J. Andrews. *Logic Programming: Operational Semantics and Proof Theory.* Cambridge University Press, 1992.

[2] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29:841–862, 1982.

[3] B. Arbab and D. M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4(4):309–329, 1987.

[4] M. Baudinet. Proving termination properties of Prolog programs: A semantic approach. *Journal of Logic Programming*, 14:1–29, 1992.

[5] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[6] P. Brisset and O. Ridoux. Continuations in λProlog. In *ICLP'93: 10th Int. Conference on Logic Programming*, Budapest, 1993.

[7] E. P. de Vink. Comparative semantics for Prolog with cut. *Science of Computer Programming*, 13(1):237–264, 1989.

[8] S. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

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

[10] P. Deransart and G. Ferrand. An operational formal definition of Prolog: A specification method an its application. *New Generation Computing*, 10:121–171, 1992.

[11] ISO Prolog Standard. http://www.logic-programming.org/prolog_std.html.

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

[13] C. Kirchner and C. Ringeissen. Rule-based constraint programming. *Fundamenta Informaticae*, 34(3):225–262, 1998.

[14] M. Kulaš. A rewriting Prolog semantics. In *CL'2000 Workshop on Verification and Computational Logic, London*, July 2000. Southampton University Tech. Report DSSE-TR-2000-6, also http://www.ecs.soton.ac.uk/~mal/vcl2000.html.

[15] T. Lindgren. A continuation-passing style for Prolog. In *ILPS'94: 11th Int. Symposium on Logic Programming*, pages 603–617, Ithaca, NY, 1994.

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

[17] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[18] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Trans. on Prog. Lang. and Systems*, 11(4):650–665, 1989.

[19] R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.

[20] M. Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995.

[21] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[22] D. A. Schmidt. On the need for a popular formal semantics. *ACM SIGPLAN Notices*, 32(1):115–116, 1997.

[23] P. Tarau and V. Dahl. Logic programming and logic grammars with binarization and first-order continuations. In *LOPSTR'94: 4th Int. Workshop on Logic Program Synthesis and Transformation, Pisa*, volume 883 of *LNCS*. Springer-Verlag, 1994.