Published in *Proc. of the CL'2000 Workshop on Verification and Computational Logic (VCL'2000)*, Southampton University Tech. Report DSSE-TR-2000-6, http://www.ecs.soton.ac.uk/~mal/vcl2000.html

A Rewriting Prolog Semantics

M. Kulaš

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

Abstract. We introduce a lean and intuitive operational semantics of a characteristic subset of Standard Prolog (the whole logic and control, database updates and solution collecting). The Prolog computation is modelled as conditional rewriting of derivation states. Due to a novel linear representation of the Prolog tree traversal, our approach is especially suitable for modelling the control flow. The semantics is fully operational and implemented in Prolog.

Table of Contents

1 Introduction

The classical view of Prolog derivation stops at the level of resolution [Llo87]. For the purposes of verification within our transformational approach to annotating full Prolog, we needed a more detailed notion of a derivation. So we came up with a novel continuation-based semantics of Prolog execution (cf. [JM84], [Sch86], [NF89], [And92], [BR93], [Pet95]). Some highlights of our approach:

- a novel linear representation of the Prolog tree traversal, enabling small and readable specifications
- closer to the spirit of logic programming: by treating if-then-else and disjunction simply and compatibly with cut, by building the Prolog tree incrementally as a partial data structure, by treating findall and database updates without "external mechanisms"
- a derivation is represented at the level of unification and backtracking
- easy finding of ancestors, thus very suitable for defining cut and catch/throw in essentially the same user-friendly way

In this paper we present the semantics in the form of (conditional) rewrite rules of the rewriting logic [Mes92], for the combined advantages of an operational model (which is in our view more readable than the usual denotational approach via lambda-calculus, offers access to intermediate states, etc.) and a lean mathematical (i. e. denotational) model.

After introducing the semantics, we give several examples, including a definition of the Byrd's semantics [Byr80] in our own.

2 Representing derivation states and answer substitutions

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 **D** is the *derivation operator*, **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

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

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 **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 **C**-operators may have an index, here depicted as $\boldsymbol{\Phi_i}$, $i = 0, .., n-1$, which will be explained later (Sec. 3.1).

Thus, G_i is the resolvent of $(G_{i-1})\boldsymbol{\sigma}_i$ with an appropriate program clause for each $i = 1, \ldots, n$ where $G_n = \wedge$ 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 to have a form showing explicitly 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

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

where $\boldsymbol{\rho_j}$ are substitutions.

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[1] to $\mathcal{S}$.

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

where a rule having (Φ) as the index of a **D** or a **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 *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 $\mathcal{S}$. The effect of 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 *StartBacktracking*($\mathcal{S}$). Therefore, we define the following:

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

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

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

For $k > 1$, if $\mathit{answer}_{k-1}(G)$ is undefined or *SP* does not terminate on $\mathit{next}_{k-1}(G)$, then $\mathit{answer}_k(G)$ is undefined. Otherwise, we set

$$\begin{array}{lll} \mathit{answer}_k(G) & =_{\mathrm{def}} & \mathit{ExtractAnswer}(\mathit{SP}(\mathit{next}_{k-1}(G))) \\ \mathit{next}_k(G) & =_{\mathrm{def}} & \mathit{StartBacktracking}(\mathit{SP}(\mathit{next}_{k-1}(G))) \end{array}$$

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 following rules.

3 Rewriting a derivation state

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 $\boldsymbol{\Pi}$ (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 this rule implements the *logical update view* [LO87], saying that the definition of a predicate shall be fixed at the time of its call.

$$\mathbf{D}[\![X,Y]\!] \;\Rightarrow\; \mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \;\text{ if } \neg\text{builtin}(X) \text{ and } \boldsymbol{\Phi} = \text{def}(X)$$

$$\mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \;\Rightarrow\; \boldsymbol{\sigma}\,\mathbf{D}[\![(R,Y)\sigma]\!]\,\mathbf{C}_{\boldsymbol{\Phi'}}[\![X,Y]\!] \;\text{ if resolve}(X,\boldsymbol{\Phi}) = <\boldsymbol{\sigma}\; R\; \boldsymbol{\Phi'}>$$

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

The second rule implements the actual resolution step. If our goal can be resolved with respect to the restriction $\boldsymbol{\Phi}$, giving a resolvent R and the most general unifier $\boldsymbol{\sigma}$, then the current goal will be replaced in the expected way.[2]

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 $\boldsymbol{\Phi}$ after removing the non-applicable clauses as well as the consumed ones so far. This leaves us with the choice points $\boldsymbol{\Phi'}$. 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. This is because now the **C**-operators must take over.

For a first illustration, we recommend Sec. 4.1.

[2] Recall that we use postfix notation for *applying a substitution*, i. e. $X\boldsymbol{\sigma}$ means that the substitution $\boldsymbol{\sigma}$ shall be applied on the term X.

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).

$$\boldsymbol{u}\,\mathbf{C}[\![X]\!] \;\Rightarrow\; \boldsymbol{u}\,\mathbf{D}[\![X]\!]$$

$$\boldsymbol{u}\,\mathbf{C}_{\Phi}[\![X]\!] \;\Rightarrow\; \boldsymbol{u}\,\mathbf{D}_{\Phi}[\![X]\!] \;\text{ if } \Phi \neq \emptyset$$

$$\boldsymbol{u}\,\mathbf{C}_{\emptyset}[\![X]\!] \;\Rightarrow\; \boldsymbol{u}\,\boldsymbol{u}$$

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 $\boldsymbol{u}$, then it shall be *activated*, i. e. turned into the current goal.

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.

$$\mathbf{D}[\![\mathit{true}, X]\!] \;\Rightarrow\; \mathbf{D}[\![X]\!]$$

$$\mathbf{D}[\![\mathit{fail}, X]\!] \;\Rightarrow\; \boldsymbol{u}$$

$$\mathbf{D}[\![\mathit{halt}, X]\!] \;\Rightarrow\; \mathbf{Halt}$$

$$\mathbf{D}[\![X, Y]\!] \;\Rightarrow\; \mathbf{Error} \;\text{ if var}(X)$$

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

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

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 [O'K90] 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. 3.9).

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

The symbol ε denotes the trivial (identity) substitution. We need it in several rules in order to preserve a 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, ε.

3.7 Once

Observe how enveloping the cut in a meta-call ensures that the cut won't encroach upon the parent of the once-goal.

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

3.8 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.

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

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

3.9 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 *To find the parent of the goal X, where the current goal is X, Y, look for the leftmost continuation whose argument does not end in X, Y (more formally: does not have a suffix unifiable with X, Y).*

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 working upon the *whole* input sequence, instead of a definite portion. Subroutines were needed in order to specify cut and catch/throw. But these are also not difficult to simulate in rewriting, due to the fact that the subroutines are actually rather simple and intuitive. 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[3] for cut.

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

Observe that we do not need a special marker for the ‘cut-parent’ (in the terminology of [BR95]), because in our approach the cut-parent is literally *the parent of the current goal starting with cut*. In the approach of [BR95] this is not necessarily so, due e. g. to the intricacies of the standard interpretation of if-then-else (Sec. 3.13, Sec. 3.14).

Another especially nice advantage of our representation of the Prolog tree is that the algorithm for cut can be easily generalized for throw.

3.10 Catch and throw

To find the right node in the Prolog tree to go to upon encountering $\mathit{throw}(\mathit{Ball})$, we have to find, according to [DEDC96], ”the closest ancestor node whose chosen predication has the form $\mathit{catch}(\mathit{Goal}, \mathit{Catcher}, \mathit{Recovergoal})$, which is still executing its Goal argument and such that a freshly renamed copy Ball' of Ball unifies with $\mathit{Catcher}$”.

What this amounts to in our representation, is: search along the chain of parents (by the suffix-criterion, Sec. 3.9) for the goal starting with $\mathit{catch}(G, B, R)$.

$$
\begin{array}{lll}
\mathbf{D}[\![\mathit{catch}(G,B,R), X]\!] & \Rightarrow & \boldsymbol{\varepsilon}\,\mathbf{D}[\![G, X]\!]\,\mathbf{C}[\![\mathit{catch}(G,B,R), X]\!] \\
\mathbf{D}[\![\mathit{throw}(B), X]\!] & \Rightarrow & \textbf{ThrowAnc(}(\mathit{throw}(B), X)\textbf{,}(\mathit{throw}(B), X)\textbf{)} \\
\textbf{ThrowAnc(}T\textbf{,}X\textbf{)}\ \mathbf{C}_{(\varPhi)}[\![C]\!] & \Rightarrow & \boldsymbol{u}\ \textbf{ThrowAnc(}T\textbf{,}X\textbf{)}\ \text{ if suffix}(C,X) \\
\textbf{ThrowAnc(}(\mathit{throw}(B), Y)\textbf{,}\,X\textbf{)}\ \mathbf{C}_{(\varPhi)}[\![C]\!] & \Rightarrow & \boldsymbol{u}\,\boldsymbol{\sigma}\mathbf{D}[\![(R,Z)\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\emptyset}[\![C]\!]\ \text{ if } \neg\text{suffix}(C,X) \\
 & & \text{and } C = (\mathit{catch}(G,B',R),Z) \text{ and unify(fresh}(B),B') = \boldsymbol{\sigma} \\
\textbf{ThrowAnc(}T\textbf{,}X\textbf{)}\ \mathbf{C}_{(\varPhi)}[\![C]\!] & \Rightarrow & \boldsymbol{u}\ \textbf{ThrowAnc(}T\textbf{,}C\textbf{)}\ \text{ otherwise} \\
\textbf{ThrowAnc(}T\textbf{,}X\textbf{)}\ \bullet & \Rightarrow & \textbf{Error}
\end{array}
$$

Here and in Sec. 3.16 we use ‘otherwise’ to avoid lengthy negations of conditions, meaning that the previous conditions or patterns do not succeed. For example, here the ‘otherwise’ rule (the last but one) is an abbreviation of:

$$
\begin{array}{l}
\textbf{ThrowAnc(}T\textbf{,}X\textbf{)}\ \mathbf{C}_{(\varPhi)}[\![C]\!] \ \Rightarrow\ \boldsymbol{u}\ \textbf{ThrowAnc(}T\textbf{,}C\textbf{)}\ \text{ if } \neg\text{suffix}(C,X) \\
\qquad\qquad\qquad \text{and } \big(C \neq (\mathit{catch}(G,B',R),Z) \text{ or unify(fresh}(B),B') = \boldsymbol{u}\big)
\end{array}
$$

In Sec. 4.2 we give a detailed example.

[3] Recall that a rule having ($\boldsymbol{\varPhi}$) 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 $\boldsymbol{\varPhi}$.

3.11 Findall

We specify findall with a (in our opinion) hitherto unattainable simplicity, due to the possibility of 'memorizing' arbitrary terms along the derivation of a goal. So, the whole idea is to leave a start-marker on the onset of deriving $\mathit{findall}(\mathit{Template}, \mathit{Enumerator}, \mathit{List})$, and after each computed answer for the enumerator to leave a further marker containing (the current instantiation of) the template. When there are no more answers, just 'rewind the tape' (i. e. go through the derivation state from the current goal up to the start-marker) and underway collect the templates memorized in the markers. That is all.

$$\ldots\textbf{StartMark}\ldots\ \textbf{Mark}(T_1)\ldots\textbf{Mark}(T_2)\ldots\textbf{Mark}(T_n)\ldots\mathbf{D}[\![\mathit{NowCollectMarks}]\!]\ldots\bullet$$

Observe that $T_1, T_2, ...T_n$ will automatically have the right instantiations, so the whole labor of the rules below consists in collecting the marks. Embedded findall goals will be treated correctly as in [CM87], actually this is the same algorithm but without need for assert/retract.

$$\mathbf{D}[\![\mathit{findall}(X,G,L),Y]\!] \;\Rightarrow\; \textbf{StartMark}\,\varepsilon\,\mathbf{D}[\![\mathit{call}(G),\ \textbf{SetMark}(X),\mathit{fail}]\!]\ \mathbf{C}[\![\ \textbf{Collect}(\mathit{findall}(X,G,L)),Y]\!]$$

$$\mathbf{D}[\![\ \textbf{SetMark}(X),Y]\!] \;\Rightarrow\; \textbf{Mark}(X1)\,\mathbf{D}[\![Y]\!]\ \text{ if fresh}(X) = X1$$

$$\mathbf{D}[\![\ \textbf{Collect}(\mathit{findall}(X,G,L)),Y]\!] \;\Rightarrow\; \textbf{SweepMarks}([])\ \textbf{Pending}(\mathbf{D}[\![\mathit{findall}(X,G,L),Y]\!])$$

$$\boldsymbol{\sigma}\ \textbf{SweepMarks}(L) \;\Rightarrow\; \textbf{SweepMarks}(L)$$

$$\boldsymbol{u}\ \textbf{SweepMarks}(L) \;\Rightarrow\; \textbf{SweepMarks}(L)$$

$$\textbf{Mark}(X)\ \ \textbf{SweepMarks}(L) \;\Rightarrow\; \textbf{SweepMarks}([X|L])$$

$$\textbf{StartMark SweepMarks}(L) \;\Rightarrow\; \textbf{Return}(L)$$

$$\textbf{Return}(L1)\ \textbf{Pending}(\mathbf{D}[\![\mathit{findall}(X,G,L),Y]\!]) \;\Rightarrow\; \boldsymbol{\sigma}\,\mathbf{D}[\![Y\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\emptyset}[\![\mathit{findall}(X,G,L),Y]\!]\ \text{ if unify}(L1,L) = \boldsymbol{\sigma}$$

$$\textbf{Return}(L1)\ \textbf{Pending}(\mathbf{D}[\![\mathit{findall}(X,G,L),Y]\!]) \;\Rightarrow\; \boldsymbol{u}\ \text{ if unify}(L1,L) = \boldsymbol{u}$$

Bagof and setof can be specified in the same way, differing from findall only in the technical issue of needing some pre-processing of the items to memorize (instead of memorizing simply the template) as well as some post-processing of the collected items.

3.12 Clause creation and destruction

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

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

$$\mathbf{D}[\![\mathit{asserta}(K),X]\!] \;\Rightarrow\; \textbf{Update}(\mathit{asserta}(K))\ \textbf{Pending}(\mathbf{D}[\![\mathit{asserta}(K),X]\!])$$

$$\mathbf{D}[\![\mathit{retract}(K),X]\!] \;\Rightarrow\; \textbf{Update}(\mathit{retract}(K))\ \textbf{Pending}(\mathbf{D}[\![\mathit{retract}(K),X]\!])$$

$$\mathbf{P}[\![\boldsymbol{\Pi}]\!]\ \textbf{Update}(\mathit{asserta}(K)) \;\Rightarrow\; \mathbf{P}[\![\boldsymbol{\Pi'}]\!]\ \textbf{Return}(\varepsilon)\ \text{ if addaclause}(K,\boldsymbol{\Pi}) = \boldsymbol{\Pi'}$$

$$\mathbf{P}[\![\boldsymbol{\Pi}]\!]\ \textbf{Update}(\mathit{retract}(K)) \;\Rightarrow\; \mathbf{P}[\![\boldsymbol{\Pi'}]\!]\ \textbf{Return}(\boldsymbol{\sigma})\ \text{ if delfirstclause}(K,\boldsymbol{\Pi}) = <\boldsymbol{\sigma}\ \boldsymbol{\Pi'}>$$

$$\mathbf{P}[\![\boldsymbol{\Pi}]\!]\ \textbf{Update}(\mathit{retract}(K)) \;\Rightarrow\; \mathbf{P}[\![\boldsymbol{\Pi}]\!]\ \textbf{Return}(\boldsymbol{u})\ \text{ if delfirstclause}(K,\boldsymbol{\Pi}) = \boldsymbol{u}$$

$$X\ \textbf{Update}(\mathit{How}) \;\Rightarrow\; \textbf{Update}(\mathit{How})\ X\ \text{ if } X \neq \mathbf{P}[\![.]\!]$$

$$\textbf{Return}(Z)\ \textbf{Pending}(\mathbf{D}[\![\mathit{asserta}(K),X]\!]) \;\Rightarrow\; \mathbf{D}[\![X]\!]$$

$$\textbf{Return}(\boldsymbol{\sigma})\ \textbf{Pending}(\mathbf{D}[\![\mathit{retract}(K),X]\!]) \;\Rightarrow\; \boldsymbol{\sigma}\,\mathbf{D}[\![X\boldsymbol{\sigma}]\!]\ \mathbf{C}[\![\mathit{retract}(K),X]\!]$$

$$\textbf{Return}(\boldsymbol{u})\ \textbf{Pending}(\mathbf{D}[\![\mathit{retract}(K),X]\!]) \;\Rightarrow\; \boldsymbol{u}$$

$$\textbf{Return}(Z)\ X \;\Rightarrow\; X\ \textbf{Return}(Z)\ \text{ if } X \neq \textbf{Pending}(.)$$

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 def(X) of a user-defined goal X.

Here we recycled the place-holding marker Pending from Sec. 3.9, and introduced a further marker Update whose purpose is to find and update the program. The subroutine-exiting marker Return (cf. Sec. 3.11, Sec. 3.9) is used to bring the resulting substitution back to the pending goal.

3.13 If-then, first draft

Specification of if-then and if-then-else of Prolog is rather difficult. Even [DEDC96] gives two different readings for if-then. According to [DEDC96], behaviour of $\mathit{If} \rightarrow \mathit{Then}$ is equivalent to $\mathit{call}(\mathit{If}), !, \mathit{Then}$, or $\mathit{If}, !, \mathit{Then}$ (page 106 op. cit.). The second one of these terms is transparent for cut within If, which if-then may not be, so we will disregard it. Let us try the first one:

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

The problem with this is, it introduces a cut, so precaution must be taken to confine it in its proper limits. Namely, in case of failure of Then, the choice points for the parent goal may not be cut off.

3.14 If-then, second draft

Therefore, we try our ‘cut insulator’ tactic:

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

But this is still not correct, because now the if-then goal is not transparent for cut *within the then-part* any more. This is probably the reason why the SICStus Prolog User’s Manual [Lab98] states about if-then-else: ”defined as if by

$$\begin{array}{l}(P \rightarrow Q; R) : -P, !, Q.\\ (P \rightarrow Q; R) : -R.\end{array}$$

except the scope of any cut in Q or R extends beyond the if-then-else construct.”

Here the problem becomes quite clear: How to achieve transparency for cut within Then or Else, and at the same time not cut too much away?

Apropos cutting too much, suppose there are no cuts in Then, but If succeeds and Then fails: the parent goal must be allowed to try alternative clauses. This would be guaranteed by the definitions above (by our rule as well as by the SICStus manual’s clauses), but they are not transparent in Then. On the other hand, the definition in the Sec. 3.13 is transparent in Then, but it cuts too much away.

3.15 If-then, correct version (with respect to cut)

So we propose to abandon the standard ‘cutty’ approach altogether, and introduce the following specification for if-then:

$$\mathbf{D}[\![(\mathit{If} \rightarrow \mathit{Then}), Y]\!] \;\Rightarrow\; \mathbf{D}[\![\mathit{once}(\mathit{If}), \mathit{Then}, Y]\!]$$

Now we have a formal definition for if-then which is obviously transparent *for cut within* Then*, but not for cut within* If, just as needed.

3.16 If-then-else

For sequential Prolog, if-then-else can be made fully compatible with disjunction, by adding a marker whose purpose is to disable the else-part after the success of the if-part. This is achieved by finding and emptying the continuation for the else-part,

$$\begin{array}{lll}
\mathbf{D}[\![(\mathit{If} \to \mathit{Then}; \mathit{Else}), Y]\!] & \Rightarrow & \mathbf{D}[\![(\mathit{once}(\mathit{If}),\ \textbf{ThisBranch}(\mathit{once}(\mathit{If}))\ , \mathit{Then}; \mathit{Else}), Y]\!] \\
\mathbf{D}[\![\ \textbf{ThisBranch}(\mathit{Pre})\ , X]\!] & \Rightarrow & \textbf{Pending(}\mathbf{D}[\![X]\!]\textbf{)}\ \textbf{The}(\mathit{Pre}\textbf{,}X) \\
\textbf{The}(\mathit{Pre}\textbf{,}\mathit{Post})\,\mathbf{C}[\![;, Y]\!] & \Rightarrow & \textbf{Return}\ \mathbf{C}_{\emptyset}[\![;, Y]\!] \ \text{ if append}(\mathit{Pre}, [\ \textbf{ThisBranch}(\mathit{Pre})\ |\mathit{Post}]) = Y \\
\textbf{The}(\mathit{Pre}\textbf{,}\mathit{Post})\,\mathbf{C}[\![X]\!] & \Rightarrow & \mathbf{C}[\![X]\!]\ \textbf{The}(\mathit{Pre}\textbf{,}\mathit{Post}) \ \text{ otherwise}
\end{array}$$

Here, if-then-else is actually a *disjunction* of the if-then and the else parts, precisely as the syntax suggests.

To understand the technical issue of searching for the continuation for the else-part, see our ‘conjunctifying’ of disjunction, Sec. 3.18.

3.17 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

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

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.)

3.18 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:

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

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. 4.4.

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 of 3.16

for if-then-else must be adapted for the enhanced syntax, namely the first rules of 3.18 and 3.16 have to be blended like this:

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

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 $\boldsymbol{\Pi}$ 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 $p(X)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

No further rules of *SP* are applicable, so this is the normal form of our initial derivation state **D**⟦p(X)⟧ • with respect to *SP*, or more formally:

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

(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 $\mathit{answer}_1(p(X))$

$$\begin{aligned}\mathit{answer}_1(p(X)) &= \mathit{ExtractAnswer}(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon}\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\boldsymbol{\Phi''}}[\![p(X)]\!]\ \bullet)\\ &= \boldsymbol{\sigma_2 \varepsilon}\,\bullet\end{aligned}$$

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 $\mathit{next}_1(p(X))$

$$\begin{aligned}\mathit{next}_1(p(X)) &= \mathit{StartBacktracking}(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon}\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\boldsymbol{\Phi''}}[\![p(X)]\!]\ \bullet)\\ &= \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon u}\mathbf{C}_\emptyset[\![p(4)]\!]\,\mathbf{C}_{\boldsymbol{\Phi''}}[\![p(X)]\!]\ \bullet\end{aligned}$$

So the sequence generated by $\mathit{next}_1(p(X))$ is the new starting state for *SP*.

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

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

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

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

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

The second answer will be given by

$$\begin{aligned}\mathit{answer}_2(p(X)) &= \mathit{ExtractAnswer}(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3}\mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet)\\ &= \boldsymbol{\sigma_3}\,\bullet\end{aligned}$$

The computation of further answers will start from the sequence

$$\begin{aligned}\mathit{next}_2(p(X)) &= \mathit{StartBacktracking}(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3}\mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet)\\ &= \boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 \varepsilon uu \sigma_3 u}\mathbf{C}_\emptyset[\![p(X)]\!]\ \bullet\end{aligned}$$

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

$$\begin{aligned}\mathit{answer}_3(p(X)) &= \mathit{ExtractAnswer}(\boldsymbol{\sigma_1 \varepsilon\varepsilon uuu \sigma_2 uu \sigma_3 uu}\,\bullet)\\ &= \boldsymbol{u}\,\bullet\end{aligned}$$

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

4.2 Catch and throw

Let our program $\boldsymbol{\Pi}$ consist of the following five clauses (adapted from [DEDC96], p. 174):

```
p.
p :- throw(b).
q :- s, catch(p, _, write(hellop)), r(c).
s.
r(X) :- throw(X).
```

We show a somewhat condensed derivation of $catch(q, C, write(helloq))$, where we skip over the applications of the first rule of 3.1, being fairly predictable.

$$\mathbf{D}[\![catch(q, C, write(helloq)]\!]$$

$$\Rightarrow\ \varepsilon\mathbf{D}[\![q]\!]\,\mathbf{C}[\![catch(q, C, write(helloq)]\!] \quad \text{by 3.10, rule 1}$$

$$\Rightarrow\ \varepsilon\varepsilon\mathbf{D}[\![s, catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, C, write(helloq)]\!] \quad \text{by 3.1, rules 1-2}$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\mathbf{D}[\![true, catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\mathbf{D}[\![catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\varepsilon\mathbf{D}[\![p, r(c)]\!]\,\mathbf{C}[\![catch(p, \_, write(hellop)), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p, ...), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\mathbf{D}[\![true, r(c)]\!]\,\mathbf{C}_{\Phi}[\![p, r(c)]\!]\,\mathbf{C}[\![catch(p, ...), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p, ...), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\mathbf{D}[\![r(c)]\!]\,\mathbf{C}_{\Phi}[\![p, r(c)]\!]\,\mathbf{C}[\![catch(p, ...), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p, ...), r(c)]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\{X/c\}\mathbf{D}[\![throw(c)]\!]\,\mathbf{C}_{\emptyset}[\![r(c)]\!]\,\mathbf{C}_{\Phi}[\![p, r(c)]\!]\,\mathbf{C}[\![catch(p...]\!]\,\mathbf{C}_{\emptyset}[\![s, catch(p...]\!]\,\mathbf{C}_{\emptyset}[\![q]\!]\,\mathbf{C}[\![catch(q, ...)]\!]$$

$$\Rightarrow\ \varepsilon\varepsilon\varepsilon\varepsilon\varepsilon\{X/c\}\boldsymbol{uuuuuu}\{C/c\}\mathbf{D}[\![write(helloq]\!]\,\mathbf{C}_{\emptyset}[\![catch(q, ...)]\!] \quad \text{by 3.10, ThrowAnc}$$

which in the next and final step prints $helloq$. The computed answer substitution is {C/c}.

Notice how, in the second line form below, the $catch(p, \_, write(hellop))$ continuation was skipped over, although its uninstantiated 'catcher' argument is unifiable with the 'ball' c, on the grounds of not belonging to the ancestor chain of the $throw(c)$ goal. The real ancestor catch goal here is the $catch(q, C, write(helloq))$. On finding it, the intervening continuations have to be emptied (including the found one) and backtracking has to be started, with the cumulative effect of physically erasing the continuations and replacing each one with an undo operator $\boldsymbol{u}$.

Observe how the suffix criterion makes it straightforward to climb the ancestor chain.

4.3 Findall

Let our program $\boldsymbol{\Pi}$ consist of the following five clauses ([DEDC96], p. 89):

```
legs(X,6) :- insect(X).
legs(X,4) :- animal(X).
legs(X,8) :- spider(X).
insect(bee).
insect(ant).
```

The derivation of $findall(X, legs(\_, X), [X, Y, Z])$ stops with the answer $\boldsymbol{u}$. For this query, [DEDC96] gives an incorrect answer substitution.
For the query *findall*(X, *legs*(_,X), *List*) we get the answer substitution $List$ = [6,6].

4.4 Repeat, disjunction, cut

Let our program $\boldsymbol{\Pi}$ 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\ \varepsilon\mathbf{D}[\![\mathit{repeat}, p(X), (X=b, !; \mathit{fail})]\!]\ \mathbf{C}_\emptyset[\![q]\!]\\
&\Rightarrow\ \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\ \varepsilon\varepsilon\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{u}\mathbf{D}[\![\mathit{fail})]\!]\ \mathbf{C}_\Phi[\![p(X), (X=b, !; \mathit{fail})]\!]\ \mathbf{C}[\![\mathit{repeat}...]\!]\ \mathbf{C}_\emptyset[\![q]\!]\\
&\Rightarrow\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\mathbf{C}_\Phi[\![p(X), (X=b, !; \mathit{fail})]\!]\ \mathbf{C}[\![\mathit{repeat}...]\!]\ \mathbf{C}_\emptyset[\![q]\!]\\
&\Rightarrow\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\mathbf{D}_\Phi[\![p(X), (X=b, !; \mathit{fail})]\!]\ \mathbf{C}[\![\mathit{repeat}...]\!]\ \mathbf{C}_\emptyset[\![q]\!]\\
&\Rightarrow\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\boldsymbol{\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\ \varepsilon\varepsilon\boldsymbol{\sigma_1}\varepsilon\boldsymbol{uu}\boldsymbol{\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 Specifying the Byrd model

A classical challenge for each new Prolog semantics is to specify the Byrd box model of Prolog [Byr80], see e. g. [BR95], [JDR00]. In our approach the Byrd model can be specified (or shall we say: incorporated, as an additional functionality) very naturally.

Looking at the rules for user-defined predicates (Sec. 3.1), the first rule obviously captures the sole entry point of a goal, so it is the right place for the call port. This is indicated by the meta-operator **CALL**(X). Note that this rule actually is also the right place for the *fail* port as well: if eventually $\mathbf{D}_\Phi[\![X, Y]\!]$ has been completed (exhausted), either because no proof for X has been found (*no*) or because all such proofs have been rejected by the ancestor goal (*no more*), we leave the **D**-operator and shall consider the alternatives i. e. continuations to the right of the **D**-operator. So this determins a proper place for the fail-port, indicated by the meta-operator **Fail**(X). But since the fail-port just *might* eventually be reached (i. e. this is only a *promise* of the fail-port), we use a 'more cautionary' (only capitalized) name for the meta-operator, as opposed to the call-port which *already* has been reached, and therefore the corresponding meta-operator will be written in uppercase.

In a similar vein to the rule for call/fail ports we arrive upon a rule for exit/redo ports. The proper starting place is obviously the second rule for user-defined predicates, because after a goal has been resolved, *if* this new goal succeeds, our original goal shall exit, indicated by the meta-operator **Exit**(X). Once the exit-port has indeed been reached, we switch to the meta-operator **EXIT**(X).

So now we have three rules capturing the essence of the Byrd model:

$$\mathbf{D}[\![X,Y]\!] \Rightarrow \mathbf{CALL}(X)\ \mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!]\ \mathbf{Fail}(X) \quad \text{if } \neg\text{system}(X) \text{ and } \boldsymbol{\Phi} = \text{def}(X)$$

$$\mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \Rightarrow \boldsymbol{\sigma}\ \mathbf{D}[\![(R,\ \mathbf{Exit}(X)\,,Y)\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\boldsymbol{\Phi'}}[\![X,Y]\!] \quad \text{if } \neg\text{meta}(X) \text{ and resolve}(X,\boldsymbol{\Phi}) = <\boldsymbol{\sigma}\ R\ \boldsymbol{\Phi'}>$$

$$\mathbf{D}[\![\,\mathbf{Exit}(X)\,,Y]\!] \Rightarrow \mathbf{EXIT}(X)\ \mathbf{D}[\![Y]\!]\ \mathbf{Redo}(X)$$

It remains to take care of two loose ends. Since a meta-operator **Exit(.)** may now appear in a goal sequence, we must make sure it won't be handled as a user-predicate. Therefore we additionally exclude meta-predicates from the rules of Sec. 3.1, with system(X) meaning that either builtin(X) or meta(X) shall hold, and meta(_) holding for the meta-operators of the system.

The second obstacle is introduction of meta-operators **Fail(.)** and **Redo(.)** in the derivation state, possibly between the $\boldsymbol{u}$ and the **D**-operator, which may not disable backtracking. Therefore we add two more rules.

It remains to add some of the old rules (covering user-defined predicates, backtracking and the builtin $true$) to the new/adapted ones, in order to show how a minimal self-contained incorporation of the Byrd model in our model looks like. Of course it wouldn't hurt to add all the rest, modulo the technical detail of bypassing **Fail(.)** / **Redo(.)** operators along the derivation state.

5.1 Byrd model, first draft

We end up with ten rules as our first draft of the Byrd model. Note however that it specifies the Byrd model *correctly and completely*, even handling the variables, which are absent from the original proposal [Byr80] or [JDR00]. So here are the rules.

$$\mathbf{D}[\![X,Y]\!] \Rightarrow \mathbf{CALL}(X)\ \mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!]\ \mathbf{Fail}(X) \quad \text{if } \neg\text{system}(X) \text{ and } \boldsymbol{\Phi} = \text{def}(X)$$

$$\mathbf{D}[\![\,\mathbf{Exit}(X)\,,Y]\!] \Rightarrow \mathbf{EXIT}(X)\ \mathbf{D}[\![Y]\!]\ \mathbf{Redo}(X)$$

$$\mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \Rightarrow \boldsymbol{\sigma}\ \mathbf{D}[\![(R,\ \mathbf{Exit}(X)\,,Y)\boldsymbol{\sigma}]\!]\ \mathbf{C}_{\boldsymbol{\Phi'}}[\![X,Y]\!] \quad \text{if } \neg\text{meta}(X) \text{ and resolve}(X,\boldsymbol{\Phi}) = <\boldsymbol{\sigma}\ R\ \boldsymbol{\Phi'}>$$

$$\mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \Rightarrow \boldsymbol{u} \quad \text{if } \neg\text{meta}(X) \text{ and resolve}(X,\boldsymbol{\Phi}) = \boldsymbol{u}$$

$$\boldsymbol{u}\,\mathbf{C}[\![X]\!] \Rightarrow \boldsymbol{u}\,\mathbf{D}[\![X]\!]$$

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

$$\boldsymbol{u}\,\mathbf{C}_{\emptyset}[\![X]\!] \Rightarrow \boldsymbol{u}\,\boldsymbol{u}$$

$$\boldsymbol{u}\ \mathbf{Fail}(X) \Rightarrow \mathbf{FAIL}(X)\ \boldsymbol{u}$$

$$\boldsymbol{u}\ \mathbf{Redo}(X) \Rightarrow \mathbf{REDO}(X)\ \boldsymbol{u}$$

$$\mathbf{D}[\![true,X]\!] \Rightarrow \mathbf{D}[\![X]\!]$$

The meta-operators **CALL(.)**, **EXIT(.)**, **FAIL(.)** and **REDO(.)** are hitherto unspecified, meaning they will stay put in the derivation state like place-holders, which might be just as well because it gives further structure to the derivation state (useful for debugging).

More general and useful would be to 'specify them away', to e. g. print the usual trace messages or do something more complex like checking assertions [Kul00]. Sec. 5.3 gives a useful application. This is the purpose of our final draft.

5.2 Byrd model, parametrized

Let us list only those rules which are now different.

$$\mathbf{D}[\![X,Y]\!] \Rightarrow \mathbf{D}_{\boldsymbol{\Phi}}[\![\,\mathbf{CALL}(X)\,,X,Y]\!]\ \mathbf{Fail}(X)\ \text{ if } \neg\text{system}(X) \text{ and } \boldsymbol{\Phi} = \text{def}(X)$$

$$\mathbf{D}[\![\,\mathbf{Exit}(X)\,,Y]\!] \Rightarrow \mathbf{D}[\![\,\mathbf{EXIT}(X)\,,Y]\!]\ \mathbf{Redo}(X)$$

$$\boldsymbol{u}\ \mathbf{Fail}(X) \Rightarrow \mathbf{D}[\![\,\mathbf{FAIL}(X)\,,\mathit{fail}]\!]$$

$$\boldsymbol{u}\ \mathbf{Redo}(X) \Rightarrow \mathbf{D}[\![\,\mathbf{REDO}(X)\,,\mathit{fail}]\!]$$

Here of course we have to additionally specify the **CALL**(.), **EXIT**(.), **FAIL**(.) and **REDO**(.). By regarding the Byrd model as 'parametrized' by these four operators, we obtain a general debugging tool in the tradition of [O'K84], see also [Kul00].

A notorious variant of this (but useful in prototyping a semantics) would be to push the four operators up into the semantic formalism:

$$\mathbf{D}[\![X,Y]\!] \Rightarrow \mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!]\ \mathbf{Fail}(X)\ \text{ if } \neg\text{system}(X) \text{ and } \boldsymbol{\Phi} = \text{def}(X) \text{ and } \mathbf{CALL}(X)$$

$$\mathbf{D}[\![\,\mathbf{Exit}(X)\,,Y]\!] \Rightarrow \mathbf{D}[\![Y]\!]\ \mathbf{Redo}(X)\ \text{ if } \mathbf{EXIT}(X)$$

$$\boldsymbol{u}\ \mathbf{Fail}(X) \Rightarrow \boldsymbol{u}\ \text{ if } \mathbf{FAIL}(X)$$

$$\boldsymbol{u}\ \mathbf{Redo}(X) \Rightarrow \boldsymbol{u}\ \text{ if } \mathbf{REDO}(X)$$

e. g. if we still do not have specified some things like output streams.

Continuation-based semantics is also used in [JDR00] to specify Byrd's box model, resulting in a similar elegant specification (but without handling of variables). He uses a a higher-order functional syntax (as in a denotational approach), making the specification arguably less readable. The intermediate states of derivation, giving further debugging information, are here not accessible.

5.3 Customization

Specifying the Byrd model in our approach makes diverse enhancements and customizations much easier.

Finite failure: The fail port is often regarded as uninteresting, capturing originally 'no (more) proofs for G' rather than 'G could not be proved'. For example in SICStus Prolog [Lab98] the fail port has been redefined to capture only the genuine 'no', the finite failure.

It is not easy to specify the finite failure of full Prolog. It would be if we had only predicates which are 'logical' (instantiation-stable, monotone) in the sense that the properties of success and (finite) failure must hold under substitutions: *if a goal succeeds (resp. fails), then any instance of it shall succeed (resp. fail) as well*. This clearly does not have to hold if var, nonvar or cut is allowed, which is the case in full Prolog. For example, a good test for a specification might be the goal *(myvar(X), myvar(a))*, with *myvar(X) :- var(X)*, which has to produce precisely one finite failure.

One[4] intuitive way to specify the finite failure is by simulating essentially the local stack, as shown in [Kul00], where we annotated the four ports of the goal in question with some database updates. The same effect we can reproduce here by instantiating the 'free' meta-operators **CALL**(.), **EXIT**(.), **FAIL**(.) and **REDO**(.) like this:

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

$$\mathbf{D}[\![\,\mathbf{REDO}(X)\,,Y]\!] \Rightarrow \mathbf{D}[\![\mathit{asserta}(\mathit{lemma}(\mathit{redo})),\,Y]\!]$$

$$\mathbf{D}[\![\,\mathbf{EXIT}(X)\,,Y]\!] \Rightarrow \mathbf{D}[\![\mathit{once}(\mathit{retract}(\mathit{lemma}(\_))),\,Y]\!]$$

$$\mathbf{D}[\![\,\mathbf{FAIL}(X)\,,Y]\!] \Rightarrow \mathbf{D}[\![\mathit{once}(\mathit{retract}(\mathit{lemma}(\mathit{What}))),(\mathit{What} = \mathit{call} \rightarrow \mathit{write}('\mathit{FinFail}' : X);\mathit{true}),\,Y]\!]$$

[4] actually, we are not aware of any other *correct* specification of finite failure of Prolog with cut or var/nonvar

Unify port: The third rule of Sec. 5.1 already specifies the *unify* port:

$$\mathbf{D}_{\boldsymbol{\Phi}}[\![X,Y]\!] \;\Rightarrow\; \mathbf{UNIFY}(X,\boldsymbol{\sigma})\;\boldsymbol{\sigma}\;\mathbf{D}[\![(R,\;\mathbf{Exit}(X)\,,Y)\boldsymbol{\sigma}]\!]\;\mathbf{C}_{\boldsymbol{\Phi}'}[\![X,Y]\!] \quad \text{if } \neg\text{meta}(X)$$

$$\text{and resolve}(X,\boldsymbol{\Phi}) = <\boldsymbol{\sigma}\;R\;\boldsymbol{\Phi}'>$$

6 Related work

There is already vast literature on the subject of assigning meaning to programs written in the pure (Horn clause logic) Prolog or various practical extensions of it. [AvE82] 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. (A notable exception being [Stä94], who managed to describe the leftmost selection in classical logic, but at a price of certain limitations; the results are not extended to a full description of depth-first evaluation [Elb99].) For example, the ubiquitous depth-first evaluation employed in actual Prolog systems is modelled by the *continuations* in the *denotational semantic* approaches of [JM84] and [NF89]. The language constructs covered include the cut operator, but more advanced control constructs like catch and throw or database predicates are not treated. Among the first to specify backtracking and cut as an elegant use of continuations was also D. Schmidt in his book [Sch86]. Our approach is also based on continuations, but their contents and handling is (to our knowledge) novel, which appears to dramatically simplify the semantics.

Whereas continuations play a central role in denotational semantics, it can also be beneficial to have first-order continuations at the source level [BR93], [TD94]. [BR93] extends the semantics of [NF89] to cover new built-ins (for λProlog) that explicitly manipulate success and failure continuations. [Lin94] uses continuation passing for an intermediate representation for compilation to an abstract machine.

Operational semantics of Horn clauses is studied in detail by Andrews in [And92]. 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 [And92] 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 and negation-as-failure.

Whereas the approaches cited above do not cover all of Prolog's built-ins, [DF92] and [BR95] explicitly aim at covering full Standard Prolog. [DF92] introduces 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, [BR95] uses abtract state machines (ASM) to "provide a primary mathematical definition of Prolog". ASMs allow for concise, high-level specifications, However, it appears that the formalism is high-level in the sense of abstract (shifting some tricky issues into the meta-level, which makes the semantics difficult to implement), rather than in the sense of intuitive appeal and practical power of Prolog. Besides, the relation to logic programming is barely discernible, the procedural issues of Prolog being defined by an even more procedural (and new) mechanism.

In our view, a Prolog semantics intended for programmers should rather adopt the spirit of logic programming, striving to handle complicated issues in a simplifying, unifying manner. The second vital issue is efficient implementation.

Acknowledgments

Special thanks for major rewriting of Sec. 2 and other valuable comments improving the presentation are due to C. Beierle, as well as for some (positive) rewievs of the related work.

References

[AIAI88] Edinburgh University A. I. Applications Institute. Edinburgh Prolog tools. ftp://sunsite.-doc.ic.ac.uk/packages/prolog-pd-software/tools.tar.Z, 1988.

[And92] J. Andrews. *Logic Programming: Operational Semantics and Proof Theory*. Cambridge University Press, 1992.

[AvE82] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29:841–862, 1982.

[Bow84] Dave Bowen. *andor.pl*. In DEC-10 Prolog Library [AIAI88]. Meta-circular interpreter maintaining extended AND/OR tree.

[BR93] P. Brisset and O. Ridoux. Continuations in λProlog. In *ICLP'93: 10th Int. Conference on Logic Programming*, Budapest, 1993.

[BR95] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

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

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 3. edition, 1987.

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

[DF92] P. Deransart and G. Ferrand. An operational formal definition of Prolog: A specification method an its application. *New Generation Computing*, 10:121–171, 1992.

[Elb99] B. Elbl. A declarative semantics for depth-first logic programs. *J. of Logic Programming*, 41:27–66, 1999.

[JDR00] E. Jahier, M. Ducassé, and O. Ridoux. Specifying Byrd's box model with a continuation semantics. In *WLPE'99, Las Cruces, NM*, volume 30, issue 4 of *Electronic Notes in Theoretical Computer Science*, 2000.

[JM84] N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *ISLP'84*, pages 281–288, Atlantic City, 1984.

[Kul00] M. Kulaš. Annotations for Prolog – A concept and runtime handling. In A. Bossi, editor, *LOPSTR'99: 9th Int. Workshop on Logic-based Program Synthesis and Transformation, Venezia*, volume 1817 of *LNCS*, pages 234–254. Springer-Verlag, 2000.

[Lab98] Intelligent Systems Laboratory. SICStus Prolog user's manual, release 3.7.1. Technical report, Swedish Institute of Computer Science, 1998. http://www.sics.se/isl/sicstus/docs/.

[Lin94] T. Lindgren. A continuation-passing style for Prolog. In *ILPS'94: 11th Int. Symposium on Logic Programming*, pages 603–617, Ithaca, NY, 1994.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2. edition, 1987.

[LO87] 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.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[NF89] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Trans. on Prog. Lang. and Systems*, 11(4):650–665, 1989.

[O'K84] Richard A. O'Keefe. *advice.pl*. In DEC-10 Prolog Library [AIAI88]. Interlisp-like advice package, 1984.

[O'K90] R. A. O'Keefe. *The Craft of Prolog*. The MIT Press, 1990.

[Pet95] M. Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995.

[Sch86] D. A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[Stä94] R. F. Stärk. The declarative semantics of the Prolog selection rule. In *LICS'94: 9th Symposium on Logic in Computer Science*, pages 252–261, Paris, 1994. IEEE Computer Society Press.

[TD94] 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.