# A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog*

T. Ströder[1], F. Emmes[1], P. Schneider-Kamp[2], J. Giesl[1], and C. Fuhs[1]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
{stroeder,emmes,giesl,fuhs}@informatik.rwth-aachen.de
[2] IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

**Abstract.** We present a new operational semantics for Prolog which covers all constructs in the corresponding ISO standard (including "non-logical" concepts like cuts, meta-programming, "all solution" predicates, dynamic predicates, and exception handling). In contrast to the classical operational semantics for logic programming, our semantics is *linear* and not based on search trees. This has the advantage that it is particularly suitable for automated program analyses such as termination and complexity analysis. We prove that our new semantics is equivalent to the ISO Prolog semantics, i.e., it computes the same answer substitutions and the derivations in both semantics have essentially the same length.

## 1 Introduction

We introduce a new *state*-based semantics for Prolog. Any query $Q$ corresponds to an initial state $s_Q$ and we define a set of *inference rules* which transform a state $s$ into another state $s'$ (denoted $s \rightsquigarrow s'$). The evaluation of $Q$ is modeled by repeatedly applying inference rules to $s_Q$ (i.e., by the derivation $s_Q \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \ldots$). Essentially, our states $s$ represent the list of those goals that still have to be proved. But in contrast to most other semantics for Prolog, our semantics is *linear* (or *local*), since each state contains all information needed for the next evaluation step. So to extend a derivation $s_0 \rightsquigarrow \ldots \rightsquigarrow s_i$, one only has to consider the last state $s_i$. Thus, even the effect of cuts and other built-in predicates becomes local.

This is in contrast to the standard semantics of Prolog (as specified in the ISO standard [11, 13]), which is defined using a *search tree* built by SLD resolution with a depth-first left-to-right strategy. To construct the next node of the tree, it is not sufficient to regard the node that was constructed last, but due to backtracking, one may have to continue with ancestor goals that occurred much "earlier" in the tree. Advanced features like cuts or exceptions require even more sophisticated analyses of the current search tree. Even worse, "all solution" predicates like findall result in several search trees and the coordination of these trees is highly non-trivial, in particular in the presence of exceptions.

We show that our linear semantics is *equivalent* to the standard ISO seman-

* Supported by DFG grant GI 274/5-3, DFG Research Training Group 1298 (*AlgoSyn*), G.I.F. grant 966-116.6, and the Danish Natural Science Research Council.

tics of Prolog. It does not only yield the same answer substitutions, but we also obtain the same *termination* behavior and even the same *complexity* (i.e., the length of the derivations in our semantics corresponds to the number of unifications performed in the standard semantics). Hence, instead of analyzing the termination or complexity of a Prolog program w.r.t. the standard semantics, one can also analyze it w.r.t. our semantics.

Compared to the ISO semantics, our semantics is much more suitable for such (possibly automated) analyses. In particular, our semantics can also be used for symbolic evaluation of *abstract* states (where the goals contain *abstract variables* representing arbitrary terms). Such abstract states can be generalized ("widened") and instantiated, and under certain conditions one may even *split up* the lists of goals in states [19, 20]. In this way, one can represent all possible evaluations of a program by a finite graph, which can then be used as the basis for e.g. termination analysis. In the standard Prolog semantics, such an abstraction of a query in a search tree would be problematic, since the remaining computation does not only depend on this query, but on the whole search tree.

In [19, 20] we already used a preliminary version of our semantics for termination analysis of a subset of Prolog containing definite logic programming and cuts. Most previous approaches for termination (or complexity [9]) analysis were restricted to definite programs. Our semantics was a key contribution to extend termination analysis to programs with cuts. The corresponding implementation in the prover AProVE resulted in the most powerful tool for automated termination analysis of logic programming so far, as shown at the *International Termination Competition*.[3] These experimental results are the main motivation for our work, since they indicate that such a semantics is indeed suitable for automated termination analysis. However, it was unclear how to extend the semantics of [19, 20] to full Prolog and how to prove that this semantics is really equivalent to the ISO semantics. These are the contributions of the current paper.

Hence, this paper forms the basis which will allow the extension of automated termination techniques to *full* Prolog. Moreover, many termination techniques can be adapted to infer upper bounds on the complexity [12, 18, 22]. Thus, the current paper is also the basis in order to adapt termination techniques such that they can be used for automated complexity analysis of full Prolog.

There exist several other alternative semantics for Prolog. However, most of them (e.g., [2, 4–8, 14, 15, 17]) only handle subsets of Prolog and it is not clear how to extend these semantics in a straightforward way to full Prolog.

Alternative semantics for *full* Prolog were proposed in [3, 10, 16]. However, these semantics seem less suitable for automated termination and complexity analysis than ours: The states used in [3] are considerably more complex than ours and it is unclear how to abstract the states of [3] for automated termination analysis as in [19, 20]. Moreover, [3] does not investigate whether their semantics also yields the same complexity as the ISO standard. The approach in [10] is close to the ISO standard and thus, it has similar drawbacks as the ISO semantics, since it also works on search trees. Finally, [16] specifies standard Prolog in

[3] See `http://www.termination-portal.org/wiki/Termination_Competition`.

rewriting logic. Similar to us, [16] uses a list representation for states. However, their approach cannot be used for complexity analysis, since their derivations can be substantially longer than the number of unifications needed to evaluate the query. Since [16] does not use explicit markers for the scope of constructs like the cut, it is also unclear how to use their approach for automated termination analysis, where one would have to abstract and to split states.

The full set of all inference rules of our semantics (for all 112 built-in predicates of ISO Prolog) can be found in [21]. Due to lack of space, in the paper we restrict ourselves to the inference rules for the most representative predicates. Sect. 2 shows the rules needed for definite logic programs. Sect. 3 extends them for predicates like the cut, negation-as-failure, and call. In Sect. 4 we handle "all solution" predicates and Sect. 5 shows how to deal with dynamic predicates like assertz and retract. Sect. 6 extends our semantics to handle exceptions (using catch and throw). Finally, Sect. 7 contains our theorems on the equivalence of our semantics to the ISO semantics. All proofs can be found in [21].

## 2 Definite Logic Programming

See e.g. [1] for the basics of logic programming. As in ISO Prolog, we do not distinguish between predicate and function symbols. For a term $t = f(t_1, \ldots, t_n)$, let $root(t) = f$. A *query* is a sequence of terms, where $\square$ denotes the empty query. A *clause* is a pair $h :\!- B$ where the *head* $h$ is a term and the *body* $B$ is a query. If $B$ is empty, then one writes just "$h$" instead of "$h :\!- \square$".[4] A *Prolog program* $\mathcal{P}$ is a finite sequence of clauses.[5]

We often denote the application of a *substitution* $\sigma$ by $t\sigma$ instead of $\sigma(t)$. A substitution $\sigma$ is the *most general unifier* (*mgu*) of $s$ and $t$ iff $s\sigma = t\sigma$ and, whenever $s\gamma = t\gamma$ for some other unifier $\gamma$, there is a $\delta$ with $X\gamma = X\sigma\delta$ for all $X \in \mathcal{V}(s) \cup \mathcal{V}(t)$.[6] As usual, "$\sigma\delta$" is the composition of $\sigma$ and $\delta$, where $X\sigma\delta = (X\sigma)\delta$. If $s$ and $t$ have no *mgu* $\sigma$, we write $mgu(s,t) = \mathit{fail}$.

A Prolog program without built-in predicates is called a *definite* logic program. Our aim is to define a *linear* operational semantics where each state contains all information needed for backtracking steps. In addition, a state also contains a list of all *answer substitutions* that were found up to now. So a state has the form $\langle G_1 \mid \ldots \mid G_n \;;\; \delta_1 \mid \ldots \mid \delta_m \rangle$ where $G_1 \mid \ldots \mid G_n$ is a sequence of goals and $\delta_1 \mid \ldots \mid \delta_m$ is a sequence of answer substitutions. We do not include the clauses from $\mathcal{P}$ in the state since they remain static during the evaluation.

Essentially, a *goal* is just a *query*, i.e., a sequence of terms. However, to compute answer substitutions, a goal $G$ is labeled by a substitution which collects

[4] In ISO Prolog, whenever an empty query $\square$ is reached, this is replaced by the built-in predicate true. However, we also allow empty queries to ease the presentation.

[5] More precisely, $\mathcal{P}$ are just the program clauses for *static* predicates. In addition to $\mathcal{P}$, a Prolog program may also contain clauses for *dynamic* predicates and *directives* to specify which predicates are dynamic. As explained in Sect. 5, these directives and the clauses for dynamic predicates are treated separately by our semantics.

[6] While the ISO standard uses unification with occurs check, our semantics could also be defined in an analogous way when using unification without occurs check.

$$\frac{\Box_\delta \mid S \;;\; A}{S \;;\; A \mid \delta}\ (\textsc{Success}) \qquad \frac{(t,Q)_\delta \mid S \;;\; A}{(t,Q)_\delta^{c_1} \mid \cdots \mid (t,Q)_\delta^{c_a} \mid S \;;\; A}\ (\textsc{Case})\ \begin{array}{l}\text{if } \mathit{defined}_{\mathcal{P}}(t) \text{ and}\\ \mathit{Slice}_{\mathcal{P}}(t) =\\ (c_1,\ldots,c_a)\end{array}$$

$$\frac{(t,Q)_\delta^{h \,:\!-\, B} \mid S \;;\; A}{(B\sigma, Q\sigma)_{\delta\sigma} \mid S \;;\; A}\ (\textsc{Eval})\ \begin{array}{l}\text{if}\\ \sigma =\\ mgu(t,h)\end{array} \qquad \frac{(t,Q)_\delta^{h \,:\!-\, B} \mid S \;;\; A}{S \;;\; A}\ (\textsc{Backtrack})\ \begin{array}{l}\text{if}\\ mgu(t,h) =\\ \mathit{fail}.\end{array}$$

**Fig. 1.** Inference Rules for Definite Logic Programs

the effects of the unifiers that were used during the evaluation up to now. So if $(t_1,\ldots,t_k)$ is a query, then a goal has the form $(t_1,\ldots,t_k)_\delta$ for a substitution $\delta$. In addition, a goal can also be labeled by a clause $c$, where the goal $(t_1,\ldots,t_k)_\delta^c$ means that the next resolution step has to be performed using the clause $c$.

The *initial state* for a query $(t_1,\ldots,t_k)$ is $\langle (t_1,\ldots,t_k)_\varnothing \;;\; \varepsilon\rangle$, i.e., the query is labeled by the identity substitution $\varnothing$ and the current list of answer substitutions is $\varepsilon$ (i.e., it is empty). This initial state can be transformed by *inference rules* repeatedly. The inference rules needed for definite logic programs are given in Fig. 1. Here, $Q$ is a query, $S$ stands for a sequence of goals, $A$ is a list of answer substitutions, and we omitted the delimiters "$\langle$" and "$\rangle$" for readability.

To illustrate these rules, we use the following program where $\mathsf{member}(t_1,t_2)$ holds whenever $t_1$ unifies with any member of the list $t_2$. Consider the query $\mathsf{member}(U,[1])$.[7] Then the corresponding initial state is $\langle \mathsf{member}(U,[1])_\varnothing \;;\; \varepsilon\rangle$.

$$\mathsf{member}(X,[X|\_]). \quad (1) \qquad\qquad \mathsf{member}(X,[\_|XS]) \;:\!-\; \mathsf{member}(X,XS). \quad (2)$$

When evaluating a goal $(t,Q)_\delta$ where $\mathit{root}(t) = p$, one tries all clauses $h :\!- B$ with $\mathit{root}(h) = p$ in the order they are given in the program. Let $\mathit{defined}_{\mathcal{P}}(t)$ indicate that $\mathit{root}(t)$ is a user-defined predicate and let $\mathit{Slice}_{\mathcal{P}}(t)$ be the list of all clauses from $\mathcal{P}$ whose head has the same root symbol as $t$. However, in the clauses returned by $\mathit{Slice}_{\mathcal{P}}(t)$, all occurring variables are renamed to fresh ones. Thus, if $\mathit{defined}_{\mathcal{P}}(t)$ and $\mathit{Slice}_{\mathcal{P}}(t) = (c_1,\ldots,c_a)$, then we use a (Case) rule which replaces the current goal $(t,Q)_\delta$ by the new list of goals $(t,Q)_\delta^{c_1} \mid \ldots \mid (t,Q)_\delta^{c_a}$. As mentioned, the label $c_i$ in such a goal means that the next resolution step has to be performed using the clause $c_i$. So in our example, $\mathsf{member}(U,[1])_\varnothing$ is replaced by the list $\mathsf{member}(U,[1])_\varnothing^{(1)'} \mid \mathsf{member}(U,[1])_\varnothing^{(2)'}$, where $(1)'$ and $(2)'$ are freshly renamed variants of the clauses (1) and (2).

To evaluate a goal $(t,Q)_\delta^{h \,:\!-\, B}$, one has to check whether there is a $\sigma = mgu(t,h)$. In this case, the (Eval) rule replaces $t$ by $B$ and $\sigma$ is applied to the whole goal. Moreover, $\sigma$ will contribute to the answer substitution, i.e., we replace $\delta$ by $\delta\sigma$. Otherwise, if $t$ and $h$ are not unifiable, then the goal $(t,Q)_\delta^{h \,:\!-\, B}$ is removed from the state and the next goal is tried (Backtrack). An empty goal $\Box_\delta$ corresponds to a successful leaf in the SLD tree. Thus, the (Success) rule removes such an empty goal and adds the substitution $\delta$ to the list $A$ of answer substitutions (we denote this by "$A \mid \delta$"). Fig. 2 shows the full evaluation of the initial state $\langle \mathsf{member}(U,[1])_\varnothing \;;\; \varepsilon\rangle$. Here, $(1)'$ and $(1)''$ (resp. $(2)'$ and $(2)''$)

[7] As usual, $[t_1,\ldots,t_n]$ abbreviates $.(t_1,.(\ldots,.(t_n,[\,])\ldots))$ and $[t \mid ts]$ stands for $.(t,ts)$.

| | |
|---|---|
| | $\mathsf{member}(U, [1])_{\varnothing}\ ;\ \varepsilon$ |
| CASE | $\mathsf{member}(U, [1])_{\varnothing}^{(1)'} \mid \mathsf{member}(U, [1])_{\varnothing}^{(2)'}\ ;\ \varepsilon$ |
| EVAL | $\square_{\{U/1,\, X'/1\}} \mid \mathsf{member}(U, [1])_{\varnothing}^{(2)'}\ ;\ \varepsilon$ |
| SUCCESS | $\mathsf{member}(U, [1])_{\varnothing}^{(2)'}\ ;\ \{U/1,\, X'/1\}$ |
| EVAL | $\mathsf{member}(U, [\,])_{\{X'/U,\, XS'/[\,]\}}\ ;\ \{U/1,\, X'/1\}$ |
| CASE | $\mathsf{member}(U, [\,])_{\{X'/U,\, XS'/[\,]\}}^{(1)''} \mid \mathsf{member}(U, [\,])_{\{X'/U,\, XS'/[\,]\}}^{(2)''}\ ;\ \{U/1,\, X'/1\}$ |
| BACKTRACK | $\mathsf{member}(U, [\,])_{\{X'/U,\, XS'/[\,]\}}^{(2)''}\ ;\ \{U/1,\, X'/1\}$ |
| BACKTRACK | $\varepsilon\ ;\ \{U/1,\, X'/1\}$ |

**Fig. 2.** Evaluation for the Query $\mathsf{member}(U, [1])$

are fresh variants of (1) (resp. (2)) that are pairwise variable disjoint. So for example, $X$ and $XS$ were renamed to $X'$ and $XS'$ in $(2)'$.

## 3 Logic and Control

In Fig. 3, we present inference rules to handle some of the most commonly used pre-defined predicates of Prolog: the cut (!), negation-as-failure (\+), the predicates call, true, and fail, and the Boolean connectives *Conn* for conjunction (',') , disjunction (';'), and implication ('->').[8] As in the ISO standard, we require that in any *clause* $h :- B$, the term $h$ and the terms in $B$ may not contain variables at *predication positions*. A position is a *predication position* iff the only function symbols that may occur above it are the Boolean connectives from *Conn*. So instead of a clause $\mathsf{q}(X) :- X$ one would have to use $\mathsf{q}(X) :- \mathsf{call}(X)$.

The effect of the cut is to remove certain backtracking possibilities. When a cut in a clause $h :- B_1, !, B_2$ with $root(h) = p$ is reached, then one does not backtrack to the remaining clauses of the predicate $p$. Moreover, the remaining backtracking possibilities for the terms in $B_1$ are also disregarded. As an example, we consider a modified member program.

$$\mathsf{member}(X, [X|\_]) \ :-\ !. \quad (3) \qquad \mathsf{member}(X, [\_|XS]) \ :-\ \mathsf{member}(X, XS). \quad (4)$$

In our semantics, the elimination of backtracking steps due to a cut is accomplished by removing goals from the state. Thus, we re-define the (CASE) rule in Fig. 3. To evaluate $p(\ldots)$, one again considers all program clauses $h :- B$ where $root(h) = p$. However, every cut in $B$ is labeled by a fresh natural number $m$. For any clause $c$, let $c[!/!_m]$ result from $c$ by replacing all (possibly labeled) cuts ! on *predication positions* by $!_m$. Moreover, we add a *scope delimiter* $?_m$ to make the end of their scope explicit. As the initial query $Q$ might also contain cuts, we also label them and construct the corresponding initial state $\langle (Q\,[!/!_0])_{\varnothing} \mid ?_0\ ;\ \varepsilon \rangle$.

In our example, consider the query $\mathsf{member}(U, [1, 1])$. Its corresponding initial state is $\langle \mathsf{member}(U, [1, 1])_{\varnothing} \mid ?_0\ ;\ \varepsilon \rangle$. Now the (CASE) rule replaces the goal

[8] The inference rules for true and the connectives from *Conn* are straightforward and thus, we only present the rule for ',' in Fig. 3. See [21] for the set of all rules.

$$\frac{(t,Q)_\delta \mid S \;;\; A}{(t,Q)_\delta^{c_1[!/!_m]} \mid \cdots \mid (t,Q)_\delta^{c_a[!/!_m]} \mid ?_m \mid S \;;\; A}\ (\textsc{Case})\ \text{if } \mathit{defined}_\mathcal{P}(t),\ \mathit{Slice}_\mathcal{P}(t) = (c_1,\ldots,c_a), \text{ and } m \text{ is fresh}$$

$$\frac{(!_m,Q)_\delta \mid S' \mid ?_m \mid S \;;\; A}{Q_\delta \mid ?_m \mid S \;;\; A}\ (\textsc{Cut}) \qquad \frac{(',' (t_1,t_2),Q)_\delta \mid S \;;\; A}{(t_1,t_2,Q)_\delta \mid S \;;\; A}\ (\textsc{Conj})$$

$$\frac{?_m \mid S \;;\; A}{S \;;\; A}\ (\textsc{Failure}) \qquad \frac{(\mathsf{call}(t),Q)_\delta \mid S \;;\; A}{(t[\mathcal{V}/\mathsf{call}(\mathcal{V}),\ !/!_m],Q)_\delta \mid ?_m \mid S \;;\; A}\ (\textsc{Call})\ \begin{array}{l}\text{if } t \notin \mathcal{V}\\ \text{and } m \text{ is}\\ \text{fresh.}\end{array}$$

$$\frac{(\mathsf{fail},Q)_\delta \mid S \;;\; A}{S \;;\; A}\ (\textsc{Fail}) \qquad \frac{(\backslash+(t),Q)_\delta \mid S \;;\; A}{(\mathsf{call}(t),!_m,\mathsf{fail})_\delta \mid Q_\delta \mid ?_m \mid S \;;\; A}\ (\textsc{Not})\ \begin{array}{l}\text{where } m \text{ is}\\ \text{fresh.}\end{array}$$

**Fig. 3.** Inference Rules for Programs with Pre-defined Predicates for Logic and Control

$\mathsf{member}(U,[1,1])_\varnothing$ by $\mathsf{member}(U,[1,1])_\varnothing^{(3)'[!/!_1]} \mid \mathsf{member}(U,[1,1])_\varnothing^{(4)'[!/!_1]} \mid ?_1$. Here, $(3)'$ is a fresh variant of the rule (3) and $(3)'[!/!_1]$ results from $(3)'$ by labeling all cuts with 1, i.e., $(3)'[!/!_1]$ is the rule $\mathsf{member}(X',[X'|\_]) :- !_1$.

Whenever a cut $!_m$ is evaluated in the current goal, the (Cut) rule removes all backtracking goals up to the delimiter $?_m$ from the state. The delimiter itself must not be removed, since the current goal might still contain more occurrences of $!_m$. So after evaluating the goal $\mathsf{member}(U,[1,1])_\varnothing^{(3)'[!/!_1]}$ to $(!_1)_{\{U/1,\,X'/1\}}$, the (Cut) rule removes all remaining goals in the list up to $?_1$.

When a predicate has been evaluated completely (i.e., when $?_m$ becomes the current goal), then this delimiter is removed. This corresponds to a failure in the evaluation, since it only occurs when all solutions have been computed. Fig. 4 shows the full evaluation of the initial state $\langle \mathsf{member}(U,[1,1])_\varnothing \mid ?_0 \;;\; \varepsilon\rangle$.

The built-in predicate call allows meta-programming. To evaluate a term $\mathsf{call}(t)$ (where $t \notin \mathcal{V}$, but $t$ may contain connectives from *Conn*), the (Call) rule replaces $\mathsf{call}(t)$ by $t[\mathcal{V}/\mathsf{call}(\mathcal{V}),\ !/!_m]$. Here, $t[\mathcal{V}/\mathsf{call}(\mathcal{V}),\ !/!_m]$ results from $t$ by replacing all variables $X$ on predication positions by $\mathsf{call}(X)$ and all (possibly labeled) cuts on predication positions by $!_m$. Moreover, a delimiter $?_m$ is added to mark the scope of the cuts in $t$.

Another simple built-in predicate is fail, whose effect is to remove the current goal. By the cut, call, and fail, we can now also handle the "negation-as-failure"

| | |
|---|---|
| | $\mathsf{member}(U,[1,1])_\varnothing \mid ?_0 \;;\; \varepsilon$ |
| Case | $\mathsf{member}(U,[1,1])_\varnothing^{(3)'[!/!_1]} \mid \mathsf{member}(U,[1,1])_\varnothing^{(4)'[!/!_1]} \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Eval | $(!_1)_{\{U/1,\,X'/1\}} \mid \mathsf{member}(U,[1,1])_\varnothing^{(4)'[!/!_1]} \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Cut | $\square_{\{U/1,\,X'/1\}} \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Success | $?_1 \mid ?_0 \;;\; \{U/1,\ X'/1\}$ |
| Failure | $?_0 \;;\; \{U/1,\ X'/1\}$ |
| Failure | $\varepsilon \;;\; \{U/1,\ X'/1\}$ |

**Fig. 4.** Evaluation for the Query $\mathsf{member}(U,[1,1])$

operator \+: the (Not) rule replaces the goal $(\backslash\!+(t), Q)_\delta$ by the list $(\mathsf{call}(t), !_m, \mathsf{fail})_\delta \mid Q_\delta \mid ?_m$. Thus, $Q_\delta$ is only executed if $\mathsf{call}(t)$ fails.

As an example, consider a program with the fact a and the rule a :− a. We regard the query $\backslash\!+(',' (\mathsf{a}, !))$. The evaluation in Fig. 5 shows that the query terminates and fails (since we do not obtain any answer substitution).

| | |
|---|---|
| | $\backslash\!+(',' (\mathsf{a}, !))_\varnothing \mid ?_0 \;;\; \varepsilon$ |
| Not | $(\mathsf{call}(',' (\mathsf{a}, !)), !_1, \mathsf{fail})_\varnothing \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Call | $(',' (\mathsf{a}, !_2), !_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Conj | $(\mathsf{a}, !_2, !_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Case | $(\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}}_\varnothing \mid (\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}\,:-\,\mathsf{a}}_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Eval | $(!_2, !_1, \mathsf{fail})_\varnothing \mid (\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}\,:-\,\mathsf{a}}_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Cut | $(!_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Cut | $\mathsf{fail}_\varnothing \mid ?_1 \mid ?_0 \;;\; \varepsilon$ |
| Fail | $?_1 \mid ?_0 \;;\; \varepsilon$ |
| Failure | $?_0 \;;\; \varepsilon$ |
| Failure | $\varepsilon \;;\; \varepsilon$ |

**Fig. 5.** Evaluation for the Query $\backslash\!+(',' (\mathsf{a}, !))$

## 4 "All Solution" Predicates

We now consider the unification predicate = and the predicates findall, bagof, and setof, which enumerate all solutions to a query. Fig. 6 gives the inference rules for = and findall (bagof and setof can be modeled in a similar way, cf. [21]).

We extend our semantics in such a way that the collection process of such "all solution" predicates is performed just like ordinary evaluation steps of a program. Moreover, we modify our concept of *states* as little as possible.

A call of $\mathsf{findall}(r, t, s)$ executes the query $\mathsf{call}(t)$. If $\sigma_1, \ldots, \sigma_n$ are the resulting answer substitutions, then finally the list $[r\sigma_1, \ldots, r\sigma_n]$ is unified with $s$.

We model this behavior by replacing a goal $(\mathsf{findall}(r, t, s), Q)_\delta$ with the list $\mathsf{call}(t) \mid \%^{r,[],s}_{Q,\delta}$. Here, $\%^{r,\ell,s}_{Q,\delta}$ is a findall-*suspension* which marks the "scope" of findall-statements, similar to the markers $?_m$ for cuts in Sect. 3. The findall-suspension fulfills two tasks: it collects all answer terms ($r$ instantiated with an

$$\frac{(\mathsf{findall}(r,t,s), Q)_\delta \mid S \;;\; A}{\mathsf{call}(t)_\varnothing \mid \%^{r,[],s}_{Q,\delta} \mid S \;;\; A}\ (\textsc{FindAll}) \qquad \frac{\%^{r,\ell,s}_{Q,\delta} \mid S \;;\; A}{(\ell{=}s, Q)_\delta \mid S \;;\; A}\ (\textsc{FoundAll})$$

$$\frac{\square_\theta \mid S' \mid \%^{r,\ell,s}_{Q,\delta} \mid S \;;\; A}{S' \mid \%^{r,\ell|r\theta,s}_{Q,\delta} \mid S \;;\; A}\ (\textsc{FindNext})\ \text{if } S' \text{ contains no findall-suspensions}$$

$$\frac{(t_1 = t_2, Q)_\delta \mid S \;;\; A}{(Q\sigma)_{\delta\sigma} \mid S \;;\; A}\ (\textsc{UnifySuccess}) \quad \text{if } \sigma = mgu(t_1, t_2)$$

$$\frac{(t_1 = t_2, Q)_\delta \mid S \;;\; A}{S \;;\; A}\ (\textsc{UnifyFail})\ \text{if } mgu(t_1,t_2) = \mathit{fail} \qquad \frac{\square_\delta \mid S \;;\; A}{S \;;\; A \mid \delta}\ (\textsc{Success})\ \text{if } S \text{ contains no findall-suspensions}$$

**Fig. 6.** Additional Inference Rules for Prolog Programs with findall

| | |
|---|---|
| | $\mathsf{findall}(U, \mathsf{member}(U, [\mathsf{1}]), L)_{\varnothing} \mid ?_0 \;;\; \varepsilon$ |
| FINDALL | $\mathsf{call}(\mathsf{member}(U, [\mathsf{1}]))_{\varnothing} \mid \%_{\square,\varnothing}^{U,[],L} \mid ?_0 \;;\; \varepsilon$ |
| CALL | $\mathsf{member}(U, [\mathsf{1}])_{\varnothing} \mid ?_1 \mid \%_{\square,\varnothing}^{U,[],L} \mid ?_0 \;;\; \varepsilon$ |
| CASE | $\mathsf{member}(U, [\mathsf{1}])_{\varnothing}^{(3)'[!/!_2]} \mid \mathsf{member}(U, [\mathsf{1}])_{\varnothing}^{(4)'[!/!_2]} \mid ?_2 \mid ?_1 \mid \%_{\square,\varnothing}^{U,[],L} \mid ?_0 \;;\; \varepsilon$ |
| EVAL | $(!_2)_{\{U/1,\, X'/1\}} \mid \mathsf{member}(U, [\mathsf{1}])_{\varnothing}^{(4)'[!/!_2]} \mid ?_2 \mid ?_1 \mid \%_{\square,\varnothing}^{U,[],L} \mid ?_0 \;;\; \varepsilon$ |
| CUT | $\square_{\{U/1,\, X'/1\}} \mid ?_2 \mid ?_1 \mid \%_{\square,\varnothing}^{U,[],L} \mid ?_0 \;;\; \varepsilon$ |
| FINDNEXT | $?_2 \mid ?_1 \mid \%_{\square,\varnothing}^{U,[\mathsf{1}],L} \mid ?_0 \;;\; \varepsilon$ |
| FAILURE | $?_1 \mid \%_{\square,\varnothing}^{U,[\mathsf{1}],L} \mid ?_0 \;;\; \varepsilon$ |
| FAILURE | $\%_{\square,\varnothing}^{U,[\mathsf{1}],L} \mid ?_0 \;;\; \varepsilon$ |
| FOUNDALL | $([\mathsf{1}]{=}L)_{\varnothing} \mid ?_0 \;;\; \varepsilon$ |
| UNIFYSUCCESS | $\square_{\{L/[\mathsf{1}]\}} \mid ?_0 \;;\; \varepsilon$ |
| SUCCESS | $?_0 \;;\; \{L/[\mathsf{1}]\}$ |
| FAILURE | $\varepsilon \;;\; \{L/[\mathsf{1}]\}$ |

**Fig. 7.** Evaluation for the Query $\mathsf{findall}(U, \mathsf{member}(U, [\mathsf{1}]), L)$

answer substitution of $t$) in its list $\ell$ and it contains all information needed to continue the execution of the program after all solutions have been found.

If a goal is evaluated to $\square_\theta$, its substitution $\theta$ would usually be added to the list of answer substitutions of the state. However, if the goals contain a findall-suspension $\%_{Q,\delta}^{r,\ell,s}$, we instead insert $r\theta$ at the end of the list of answers $\ell$ using the (FINDNEXT) rule (denoted by "$\ell \mid r\theta$").[9] To avoid overlapping inference rules, we modify the (SUCCESS) rule such that it is only applicable if (FINDNEXT) is not.

When $\mathsf{call}(t)$ has been fully evaluated, the first element of the list of goals is a findall-suspension $\%_{Q,\delta}^{r,\ell,s}$. Before continuing the evaluation of $Q$, we unify the list of collected solutions $\ell$ with the expected list $s$ (using the built-in predicate $=$).

As an example, for the Prolog program defined by the clauses (3) and (4), an evaluation of the query $\mathsf{findall}(U, \mathsf{member}(U, [\mathsf{1}]), L)$ is given in Fig. 7.

## 5 Dynamic Predicates

Now we also consider built-in predicates which modify the program clauses for some predicate p at runtime. This is only possible for "new" predicates which were not defined in the program and for predicates where the program contains a dynamic directive before their first clause (e.g., ":− dynamic p/1"). Thus, we consider a program to consist of two parts: a static part $\mathcal{P}$ containing all program clauses for static predicates and a dynamic part, which can be modified at runtime and initially contains all program clauses for dynamic predicates.

Therefore, we extend our states by a list $\mathcal{D}$ which stores all clauses of dynamic predicates, where each of these clauses is labeled by a natural number. We now denote a state as $\langle S \;;\; \mathcal{D} \;;\; A \rangle$ where $S$ is a list of goals and $A$ is a list of answer

[9] As there may be nested findall calls, we use the first findall-suspension in the list.

$$\frac{(t,Q)_\delta \mid S \;;\; \mathcal{D} \;;\; A}{(t,Q)_\delta^{c_1[!/!_m]} \mid \cdots \mid (t,Q)_\delta^{c_a[!/!_m]} \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{Case})\ \begin{array}{l}\text{if } \mathit{defined}_\mathcal{P}(t),\\ \mathit{Slice}_{(\mathcal{P}\mid\overline{\mathcal{D}})}(t) = (c_1,\ldots,c_a),\\ \overline{\mathcal{D}} \text{ is } \mathcal{D} \text{ without clause labels,}\\ \text{and } m \text{ is fresh}\end{array}$$

$$\frac{(\mathsf{asserta}(c),Q)_\delta \mid S \;;\; \mathcal{D} \;;\; A}{Q_\delta \mid S \;;\; (c,m) \mid \mathcal{D} \;;\; A}\ (\textsc{AssA})\ \begin{array}{l}\text{if } m \in \mathbb{N}\\ \text{is fresh}\end{array} \qquad \frac{(\mathsf{assertz}(c),Q)_\delta \mid S \;;\; \mathcal{D} \;;\; A}{Q_\delta \mid S \;;\; \mathcal{D} \mid (c,m) \;;\; A}\ (\textsc{AssZ})\ \begin{array}{l}\text{if } m \in \mathbb{N}\\ \text{is fresh}\end{array}$$

$$\frac{(\mathsf{retract}(c),Q)_\delta \mid S \;;\; \mathcal{D} \;;\; A}{:\!\neq_{Q,\delta}^{c,(c_1,m_1)} \mid \cdots \mid :\!\neq_{Q,\delta}^{c,(c_a,m_a)} \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{Retract})\ \begin{array}{l}\text{if } \mathit{Slice}_\mathcal{D}(c) =\\ ((c_1,m_1),\ldots,(c_a,m_a))\end{array}$$

$$\frac{:\!\neq_{Q,\delta}^{c,(c',m)} \mid S \;;\; \mathcal{D} \;;\; A}{(Q\sigma)_{\delta\sigma} \mid S \;;\; \mathcal{D} \setminus (c',m) \;;\; A}\ (\textsc{RetSuc})\ \begin{array}{l}\text{if } \sigma =\\ mgu(c,c')\end{array} \qquad \frac{:\!\neq_{Q,\delta}^{c,(c',m)} \mid S \;;\; \mathcal{D} \;;\; A}{S \;;\; \mathcal{D} \;;\; A}\ (\textsc{RetFail})\ \begin{array}{l}\text{if}\\ mgu(c,c')\\ = \mathit{fail}\end{array}$$

**Fig. 8.** Additional Inference Rules for Prolog Programs with Dynamic Predicates

substitutions. The inference rules for the built-in predicates asserta, assertz, and retract in Fig. 8 modify the list $\mathcal{D}$.[10] Of course, the (Case) rule also needs to be adapted to take the clauses from $\mathcal{D}$ into account (here, "$\mathcal{P} \mid \overline{\mathcal{D}}$" stands for appending the lists $\mathcal{P}$ and $\overline{\mathcal{D}}$). All other previous inference rules do not depend on the new component $\mathcal{D}$ of the states.

For a clause[11] $c$, the effect of $\mathsf{asserta}(c)$ resp. $\mathsf{assertz}(c)$ is modeled by inserting $(c,m)$ at the beginning resp. the end of the list $\mathcal{D}$, where $m$ is a fresh number, cf. the rules (AssA) and (AssZ). The labels in $\mathcal{D}$ are needed to uniquely identify each clause as demonstrated by the following query for a dynamic predicate p.

$$\mathsf{assertz(p(a))},\ \mathsf{assertz(p(b))},\ \mathsf{retract(p}(X)\mathsf{)},\ \underbrace{X\mathsf{= a},\ \mathsf{retract(p(b))},\ \mathsf{assertz(p(b))},\ \mathsf{fail}}_{Q}$$

So first the two clauses p(a) and p(b) are asserted, i.e., $\mathcal{D}$ contains $(\mathsf{p(a)},1)$ and $(\mathsf{p(b)},2)$. When $\mathsf{retract(p}(X)\mathsf{)}$ is executed, one collects all p-clauses from $\mathcal{D}$, since these are the only clauses which might be removed by this retract-statement.

To this end, we extend the function *Slice* such that $\mathit{Slice}_\mathcal{D}(c)$ returns fresh variants of all labeled clauses $c'$ from $\mathcal{D}$ where $\mathit{root}(\mathit{head}(c)) = \mathit{root}(\mathit{head}(c'))$. An execution of $(\mathsf{retract}(c),Q)_\delta$ then creates a new *retract marker* for every clause in $\mathit{Slice}_\mathcal{D}(c) = ((c_1,m_1),\ldots,(c_a,m_a))$, cf. the (Retract) inference rule in Fig. 8. Such a retract marker $:\!\neq_{Q,\delta}^{c,(c_i,m_i)}$ denotes that the clause with label $m_i$ should be removed from $\mathcal{D}$ if $c$ unifies with $c_i$ by some mgu $\sigma$. Moreover, then the computation continues with the goal $(Q\sigma)_{\delta\sigma}$, cf. (RetSuc). If $c$ does not unify with $c_i$, then the retract marker is simply dropped by the rule (RetFail).

So in our example, we create the two retract markers $:\!\neq_{Q,\varnothing}^{\mathsf{p}(X),(\mathsf{p(a)},1)}$ and $:\!\neq_{Q,\varnothing}^{\mathsf{p}(X),(\mathsf{p(b)},2)}$, where $Q$ are the last four terms of the query. Since $\mathsf{p}(X)$ unifies

[10] The inference rules for the related predicate abolish are analogous, cf. [21].

[11] For $\mathsf{asserta}(c)$, $\mathsf{assertz}(c)$, and $\mathsf{retract}(c)$, we require that the body of the clause $c$ may not be empty (i.e., instead of a fact $\mathsf{p}(X)$ one would have to use $\mathsf{p}(X) :\!-\mathsf{true}$). Moreover, $c$ may not have variables on predication positions.

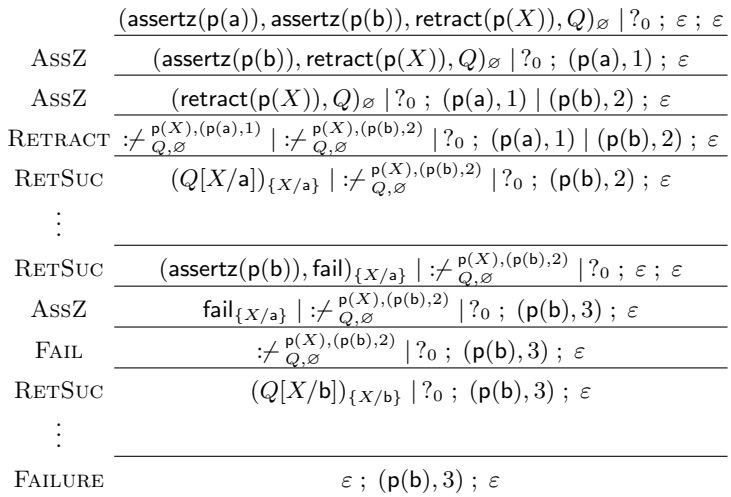


**Fig. 9.** Evaluation for a Query using assertz and retract

with p(a), the first clause (p(a), 1) is retracted from $\mathcal{D}$. Due to the unifier $\{X/\mathsf{a}\}$, the term $(X{=}\mathsf{a})[X/\mathsf{a}]$ is satisfied. Hence, retract(p(b)) and assertz(p(b)) are executed, i.e., the clause (p(b), 2) is removed from $\mathcal{D}$ and a new clause (p(b), 3) is added to $\mathcal{D}$. When backtracking due to the term fail at the end of the query, the execution of retract(p($X$)) is again successful, i.e., the retraction described by the marker $:\neq^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing}$ succeeds since p($X$) also unifies with the clause (p(b), 2). However, this retract-statement does not modify $\mathcal{D}$ anymore, since (p(b), 2) is no longer contained in $\mathcal{D}$. Due to the unifier $\{X/\mathsf{b}\}$, the next term $(X{=}\,\mathsf{a})[X/\mathsf{b}]$ is not satisfiable and the whole query fails. However, then $\mathcal{D}$ still contains (p(b), 3). Hence, afterwards a query like p($X$) would yield the answer substitution $\{X/\mathsf{b}\}$. See Fig. 9 for the evaluation of this example using our inference rules.

## 6 Exception Handling

Prolog provides an *exception handling mechanism* by means of two built-in predicates throw and catch. The unary predicate throw is used to "throw" exception terms and the predicate catch can react on thrown exceptions.

When reaching a term catch($t, c, r$), the term $t$ is called. During this call, an exception term $e$ might be thrown. If $e$ and $c$ unify with the mgu $\sigma$, the recover term $r$ is instantiated by $\sigma$ and called. Otherwise, the effect of the catch-call is the same as a call to throw($e$). If no exception is thrown during the execution of call($t$), the catch has no other effect than this call.

To model the behavior of catch and throw, we augment each goal in our states by context information for every catch-term that led to this goal. Such a catch-*context* is a 5-tuple $(m, c, r, Q, \delta)$, consisting of a natural number $m$ which marks the scope of the corresponding catch-term, a catcher term $c$ describing which exception terms to catch, a recover term $r$ which is evaluated in case of a caught

$$\frac{(\mathsf{catch}(t,c,r),Q)_{\delta,C} \mid S \;;\; \mathcal{D} \;;\; A}{\mathsf{call}(t)_{\varnothing,\, C|(m,c,r,Q,\delta)} \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{Catch})\ \text{where } m \text{ is fresh}$$

$$\frac{(\mathsf{throw}(e),Q)_{\theta,\, C|(m,c,r,Q',\delta)} \mid S' \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}{(\mathsf{call}(r\sigma),Q'\sigma)_{\delta\sigma,\, C} \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{ThrowSuccess})\ \begin{array}{l}\text{if } e \notin \mathcal{V} \text{ and } \sigma = \\ mgu(c,e') \text{ for a} \\ \text{fresh variant } e' \text{ of } e\end{array}$$

$$\frac{(\mathsf{throw}(e),Q)_{\theta,\, C|(m,c,r,Q',\delta)} \mid S' \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}{(\mathsf{throw}(e),Q)_{\theta,\, C} \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{ThrowNext})\ \begin{array}{l}\text{if } e \notin \mathcal{V} \text{ and} \\ mgu(c,e') = \mathit{fail} \\ \text{for a fresh variant} \\ e' \text{ of } e\end{array}$$

$$\frac{(\mathsf{throw}(e),Q)_{\theta,\varepsilon} \,|\, S\,;\,\mathcal{D};\,A}{\mathsf{ERROR}}\ (\textsc{ThrowErr})\ \text{if } e \notin \mathcal{V} \qquad \frac{\square_{\theta,\varepsilon} \,|\, S\,;\,\mathcal{D};\,A}{S\,;\,\mathcal{D};\,A\,|\,\theta}\ (\textsc{Success})\ \begin{array}{l}\text{if } S \text{ contains} \\ \text{no } \mathsf{findall}\text{-} \\ \text{suspensions}\end{array}$$

$$\frac{\square_{\theta,\, C|(m,c,r,Q,\delta)} \mid S' \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}{(Q\theta)_{\delta\theta,\, C} \mid S' \mid ?_m \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{CatchNext})\ \begin{array}{l}\text{if } S' \text{ contains no} \\ \mathsf{findall}\text{-suspensions}\end{array}$$

$$\frac{\square_{\theta,C} \mid S' \mid \%^{r,\ell,s}_{Q',\delta',C'} \mid S \;;\; \mathcal{D} \;;\; A}{S' \mid \%^{r,\ell|r\theta,s}_{Q',\delta',C'} \mid S \;;\; \mathcal{D} \;;\; A}\ (\textsc{FindNext})\ \begin{array}{l}\text{if } S' \text{ contains no } \mathsf{findall}\text{-suspensions and} \\ (\, C \text{ is either empty or else its last element} \\ \text{is } (m,c,r,Q,\delta) \text{ and } S' \text{ contains no } ?_m\,)\end{array}$$

**Fig. 10.** Additional Inference Rules for Prolog Programs with Error Handling

exception, as well as a query $Q$ and a substitution $\delta$ describing the remainder of the goal after the catch-term. In general, we denote a list of catch-contexts by $C$ and write $Q_{\delta,C}$ for a goal with the query $Q$ and the annotations $\delta$ and $C$.

To evaluate $(\mathsf{catch}(t,c,r),Q)_{\delta,C}$, we append the catch-context $(m,c,r,Q,\delta)$ (where $m$ is a fresh number) to $C$ (denoted by "$C \mid (m,c,r,Q,\delta)$") and replace the catch-term by $\mathsf{call}(t)$, cf. (Catch) in Fig. 10. To identify the part of the list of goals that is caused by the evaluation of this call, we add a scope marker $?_m$.

When a goal $(\mathsf{throw}(e),Q)_{\theta,\, C|(m,c,r,Q',\delta)}$ is reached, we drop all goals up to the marker $?_m$. If $c$ unifies with a fresh variant $e'$ of $e$ using an mgu $\sigma$, we replace the current goal by the instantiated recover goal $(\mathsf{call}(r\sigma),Q'\sigma)_{\delta\sigma,\, C}$ using the rule (ThrowSuccess). Otherwise, in the rule (ThrowNext), we just drop the last catch-context and continue with the goal $(\mathsf{throw}(e),Q)_{\theta,C}$. If an exception is thrown without a catch-context, then this corresponds to a program error. To this end, we extend the set of states by an additional element ERROR.

Since we extended goals by a list of catch-contexts, we also need to adapt all previous inference rules slightly. Except for (Success) and (FindNext), this is straightforward[12] since the previous rules neither use nor modify the catch-contexts. As catch-contexts can be converted into goals, findall-suspensions % and retract-markers $:\neq$ have to be annotated with lists of catch-contexts, too.

An interesting aspect is the interplay of nested catch- and findall-calls. When

[12] However, several built-in predicates (e.g., call and findall) impose "error conditions". If their arguments do not have the required form, an exception is thrown. Thus, the rules for these predicates must also be extended appropriately, cf. [21].

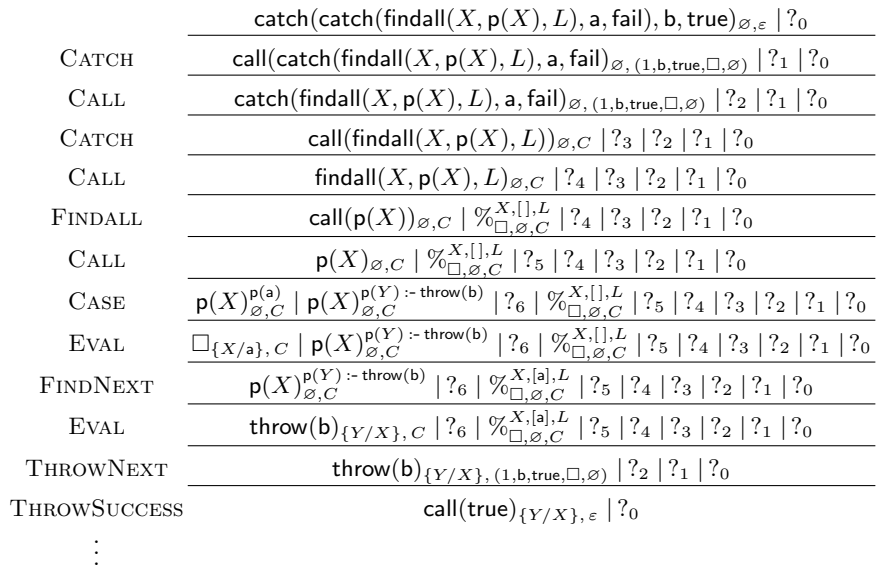


**Fig. 11.** Evaluation for a Query of Nested catch- and findall-Calls

reaching a goal $\square_{\theta,\,C|(m,c,r,Q,\delta)}$ which results from the evaluation of a catch-term, it is not necessarily correct to continue the evaluation with the goal $(Q\theta)_{\delta\theta,\,C}$ as in the rule (CatchNext). This is because the evaluation of the catch-term may have led to a findall-call and the current "success" goal $\square_{\theta,\,C|(m,c,r,Q,\delta)}$ resulted from this findall-call. Then one first has to compute the remaining solutions to this findall-call and one has to keep the catch-context $(m, c, r, Q, \delta)$ since these computations may still lead to exceptions that have to be caught by this context. Thus, then we only add the computed answer substitution $\theta$ to its corresponding findall-suspension, cf. the modified (FindNext) rule.

For the program with the fact $\mathsf{p}(\mathsf{a})$ and the rule $\mathsf{p}(Y) :- \mathsf{throw}(\mathsf{b})$, an evaluation of a query with catch and findall is given in Fig. 11. Here, the clauses $\mathcal{D}$ for dynamic predicates and the list $A$ of answer substitutions were omitted for readability. Moreover, $C$ stands for the list $(1, \mathsf{b}, \mathsf{true}, \square, \varnothing) \mid (3, \mathsf{a}, \mathsf{fail}, \square, \varnothing)$.

## 7 Equivalence to the ISO Semantics

In this section, we formally define our new semantics for Prolog and show that it is equivalent to the semantics defined in the ISO standard [11, 13]. All definitions and theorems refer to the *full* set of inference rules (handling full Prolog). As mentioned, all inference rules and all proofs can be found in [21].

**Theorem 1 ("Mutual Exclusion" of Inference Rules).** *For each state, there is at most one inference rule applicable and the result of applying this rule is unique up to renaming of variables and of fresh numbers used for markers.*

Let $s_0 \rightsquigarrow s_1$ denote that the state $s_0$ was transformed to the state $s_1$ by one of our inference rules. Any finite or infinite sequence $s_0 \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \ldots$ is

called a *derivation* of $s_0$. Thm. 1 implies that any state has a unique maximal derivation (which may be infinite). Now we can define our semantics for Prolog.

**Definition 2 (Linear Semantics for Prolog).** *Consider a Prolog program with the clauses $\mathcal{P}$ for static predicates and the clauses $\overline{\mathcal{D}}$ for dynamic predicates. Let $\mathcal{D}$ result from $\overline{\mathcal{D}}$ by labeling each clause in $\overline{\mathcal{D}}$ by a fresh natural number. Let $Q$ be a query and let $s_Q = \langle S_Q; \mathcal{D}; \varepsilon \rangle$ be the* corresponding initial state*, where $S_Q = (Q[!/!_0])_{\varnothing,\varepsilon} \mid ?_0$.*

*(a) We say that the execution of $Q$ has* length $\ell \in \mathbb{N} \cup \{\infty\}$ *iff the maximal derivation of $s_Q$ has length $\ell$. In particular, $Q$ is called* terminating *iff $\ell \neq \infty$.*
*(b) We say that $Q$ leads to a* program error *iff the maximal derivation of $s_Q$ is finite and ends with the state* ERROR*.*
*(c) We say that $Q$ leads to the (finite or infinite) list of* answer substitutions *$A$ iff either the maximal derivation of $s_Q$ is finite and ends with a state $\langle \varepsilon; \mathcal{D}'; A \rangle$, or the maximal derivation of $s_Q$ is infinite and for every finite prefix $A'$ of $A$, there exists some $S$ and $\mathcal{D}'$ with $s_Q \rightsquigarrow^* \langle S; \mathcal{D}', A' \rangle$. As usual, $\rightsquigarrow^*$ denotes the transitive and reflexive closure of $\rightsquigarrow$.*

In contrast to Def. 2, the ISO standard [11, 13] defines the semantics of Prolog using search trees. These search trees are constructed by a depth-first search from left to right, where of course one avoids the construction of parts of the tree that are not needed (e.g., because of cuts). In the ISO semantics, we have the following for a Prolog program $\mathcal{P}$ and a query $Q$:[13]

(a) The execution of $Q$ has *length* $k \in \mathbb{N} \cup \{\infty\}$ iff $k$ unifications are needed to construct the search tree (where the execution of a built-in predicate also counts as at least one unification step).[14] Of course, here every unification attempt is counted, no matter whether it succeeds or not. So in the program with the fact p(a), the execution of the query p(b) has length 1, since there is one (failing) unification attempt.
(b) $Q$ leads to a *program error* iff during the construction of the search tree one reaches a goal $(\mathsf{throw}(e), Q)$ and the thrown exception is not caught.
(c) $Q$ leads to the list of *answer substitutions* $A$ iff $Q$ does not lead to a program error and $A$ is the list of answer substitutions obtained when traversing the (possibly infinite) search tree by depth-first search from left to right.

Thm. 3 (a) shows that our semantics and the ISO semantics result in the same termination behavior. Moreover, the computations according to the ISO semantics and our maximal derivations have the same length up to a constant factor. Thus, our semantics can be used for termination and complexity analysis of Prolog. Thm. 3 (b) states that our semantics and the ISO semantics lead to the same program errors and in (c), we show that the two semantics compute

[13] See [21] for a more formal definition.
[14] In other words, even for built-in predicates $p$, the evaluation of an atom $p(t_1, \ldots, t_n)$ counts as at least one unification step. For example, this is needed to ensure that the execution of queries like "repeat, fail" has length $\infty$.

the same answer substitutions (up to variable renaming).[15]

**Theorem 3 (Equivalence of Our Semantics and the ISO Semantics).** *Consider a a Prolog program and a query $Q$.*

*(a) Let $\ell$ be the* length *of $Q$'s execution according to our semantics in Def. 2 and let $k$ be the* length *of $Q$'s execution according to the ISO semantics. Then we have $k \leq \ell \leq 3 \cdot k + 1$. So in particular we also obtain $\ell = \infty$ iff $k = \infty$ (i.e., the two semantics have the same termination behavior).*

*(b) $Q$ leads to a* program error *according to our semantics in Def. 2 iff $Q$ leads to a* program error *according to the ISO semantics.*

*(c) $Q$ leads to a (finite or infinite) list of* answer substitutions $\delta_0, \delta_1, \ldots$ *according to our semantics in Def. 2 iff $Q$ leads to a list of* answer substitutions $\theta_0, \theta_1, \ldots$ *according to the ISO semantics, where the two lists have the same length $n \in \mathbb{N} \cup \{\infty\}$ and for each $i < n$, there exists a variable renaming $\tau_i$ such that for all variables $X$ in the query $Q$, we have $X\theta_i = X\delta_i\,\tau_i$.*

To see why we do not have $\ell = k$ in Thm. 3(a), consider again the program with the fact $\mathsf{p(a)}$ and the query $\mathsf{p(b)}$. While the ISO semantics only needs $k = 1$ unification attempt, our semantics uses 3 steps to model the failure of this proof. Moreover, in the end we need one additional step to remove the marker $?_0$ constructed in the initial state. The evaluation is shown in Fig. 12, where we omitted the catch-contexts and the components for dynamic predicates and answer substitutions for readability. So in this example, we have $\ell = 3 \cdot k + 1 = 4$.

$$\begin{array}{rc} & (\mathsf{p(b)})_{\varnothing} \mid ?_0 \\ \hline \textsc{Case} & (\mathsf{p(b)})_{\varnothing}^{\mathsf{p(a)}} \mid ?_1 \mid ?_0 \\ \hline \textsc{Backtrack} & ?_1 \mid ?_0 \\ \hline \textsc{Failure} & ?_0 \\ \hline \textsc{Failure} & \varepsilon \end{array}$$

**Fig. 12.** Evaluation for $\mathsf{p(b)}$

## 8 Conclusion

We have presented a new operational semantics for full Prolog (as defined in the corresponding ISO standard [11, 13]) including the cut, "all solution" predicates like findall, dynamic predicates, and exception handling. Our semantics is *modular* (i.e., easy to adapt to subsets of Prolog) and *linear* resp. *local* (i.e., derivations are lists instead of trees and even the cut and exceptions are local operations where the next state in a derivation only depends on the previous state).

We have proved that our semantics is equivalent to the semantics based on search trees defined in the ISO standard w.r.t. both termination behavior and computed answer substitutions. Furthermore, the number of derivation steps in our semantics is equal to the number of unifications needed for the ISO semantics (up to a constant factor). Hence, our semantics is suitable for (possibly automated) analysis of Prolog programs, for example for static analysis of termination and complexity using an abstraction of the states in our semantics as in [19, 20].

In [19, 20], we already successfully used a subset of our new semantics for automated termination analysis of definite logic programs with cuts. In future work, we will extend termination analysis to deal with all our inference rules in

[15] Moreover, the semantics are also equivalent w.r.t. the side effects of a program (like the changes of the dynamic clauses, input and output, etc.).

order to handle full Prolog as well as to use the new semantics for asymptotic worst-case complexity analysis. We further plan to investigate uses of our semantics for debugging and tracing applications exploiting linearity and locality.

## References


1. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
2. B. Arbab and D. M. Berry. Operational and denotational semantics of Prolog. *Journal of Logic Programming*, 4:309–329, 1987.
3. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
4. S. Cerrito. A linear semantics for allowed logic programs. In *LICS '90*, pages 219–227. IEEE Press, 1990.
5. M. H. M. Cheng, R. N. Horspool, M. R. Levy, and M. H. van Emden. Compositional operational semantics for Prolog programs. *New Generat. Comp.*, 10:315–328, 1992.
6. A. de Bruin and E. P. de Vink. Continuation semantics for Prolog with cut. In *TAPSOFT '89*, LNCS 351, pages 178–192, 1989.
7. E. P. de Vink. Comparative semantics for Prolog with cut. *Science of Computer Programming*, 13:237–264, 1990.
8. S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.
9. S. K. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15:826–875, 1993.
10. P. Deransart and G. Ferrand. An operational formal definition of Prolog: a specification method and its application. *New Generation Computing*, 10:121–171, 1992.
11. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, 1996.
12. N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *IJCAR '08*, LNAI 5195, pages 364–379, 2008.
13. ISO/IEC 13211-1. *Information technology - Programming languages - Prolog*. 1995.
14. J. Jeavons. An alternative linear semantics for allowed logic programs. *Annals of Pure and Applied Logic*, 84(1):3–16, 1997.
15. N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *SLP '84*, pages 281–288. IEEE Press, 1984.
16. M. Kulaš and C. Beierle. Defining standard Prolog in rewriting logic. In *WRLA '00*, ENTCS 36, 2001.
17. T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11:650–665, 1989.
18. L. Noschinski, F. Emmes, J. Giesl. The dependency pair framework for automated complexity analysis of term rewrite systems. In *CADE '11*, LNAI, 2011. To appear.
19. P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated termination analysis for logic programs with cut. In *ICLP '10, Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010.
20. T. Ströder, P. Schneider-Kamp, J. Giesl. Dependency Triples for Improving Termination Analysis of Logic Programs with Cut. In *LOPSTR '10*, LNCS 6564, pages 184–199, 2011.
21. T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A linear operational semantics for termination and complexity analysis of ISO Prolog. Technical Report AIB-2011-08, RWTH Aachen, 2011. Available from `http://aib.informatik.rwth-aachen.de/`.
22. H. Zankl and M. Korp. Modular complexity analysis via relative complexity. In *RTA '10*, LIPIcs 6, pages 385–400, 2010.