RWTH Aachen

Department of Computer Science
*Technical Report*

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

Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, and Carsten Fuhs

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2011-08

RWTH Aachen · Department of Computer Science · July 2011

# 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 $s \leadsto s'$). The evaluation of $Q$ is modeled by repeatedly applying inference rules to $s_Q$ (i.e., by the derivation $s_Q \leadsto s_1 \leadsto s_2 \leadsto \ldots$). 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 $s_0 \leadsto \ldots \leadsto s_i$, 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, 14]), 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 (*Algo-Syn*), 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 [20, 21]. 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 [20, 21] 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 [20, 21] 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, 19, 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, 15, 16, 18]) 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, 17]. 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 [20, 21]. 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, [17] specifies standard Prolog in

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

rewriting logic. Similar to us, [17] 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 [17] 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 Appendix B. In the main part of 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 Appendix 7.

## 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 $mgu(s,t) = fail$.

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.

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

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

**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 $\langle \mathsf{member}(U, [1])_\varnothing\ ;\ \varepsilon \rangle$.

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

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 $\sigma = \mathit{mgu}(t, h)$. 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 | $\mathsf{member}(U,[1])^{(1)'}_{\varnothing} \mid \mathsf{member}(U,[1])^{(2)'}_{\varnothing}\ ;\ \varepsilon$ |
| EVAL | $\Box_{\{U/1,\ X'/1\}} \mid \mathsf{member}(U,[1])^{(2)'}_{\varnothing}\ ;\ \varepsilon$ |
| SUCCESS | $\mathsf{member}(U,[1])^{(2)'}_{\varnothing}\ ;\ \{U/1,\ X'/1\}$ |
| EVAL | $\mathsf{member}(U,[\,])_{\{X'/U,\ XS'/[\,]\}}\ ;\ \{U/1,\ X'/1\}$ |
| CASE | $\mathsf{member}(U,[\,])^{(1)''}_{\{X'/U,\ XS'/[\,]\}} \mid \mathsf{member}(U,[\,])^{(2)''}_{\{X'/U,\ XS'/[\,]\}}\ ;\ \{U/1,\ X'/1\}$ |
| BACKTRACK | $\mathsf{member}(U,[\,])^{(2)''}_{\{X'/U,\ XS'/[\,]\}}\ ;\ \{U/1,\ X'/1\}$ |
| 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 App. B for the set of all rules.

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

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

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

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

**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 $\mathsf{member}(X',[X'|\_]) \mathrel{:-} !_1$.

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 $\mathsf{call}$ 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 $\mathsf{fail}$, whose effect is to remove the current goal. By the cut, $\mathsf{call}$, and $\mathsf{fail}$, we can now also handle the "negation-as-failure"

$$\begin{array}{r|c|}
 & \mathsf{member}(U,[1,1])_\varnothing \mid ?_0 \;;\; \varepsilon \\ \cline{2-2}
\textsc{Case} & \mathsf{member}(U,[1,1])_\varnothing^{(3)'[!/!_1]} \mid \mathsf{member}(U,[1,1])_\varnothing^{(4)'[!/!_1]} \mid ?_1 \mid ?_0 \;;\; \varepsilon \\ \cline{2-2}
\textsc{Eval} & (!_1)_{\{U/1,\,X'/1\}} \mid \mathsf{member}(U,[1,1])_\varnothing^{(4)'[!/!_1]} \mid ?_1 \mid ?_0 \;;\; \varepsilon \\ \cline{2-2}
\textsc{Cut} & \square_{\{U/1,\,X'/1\}} \mid ?_1 \mid ?_0 \;;\; \varepsilon \\ \cline{2-2}
\textsc{Success} & ?_1 \mid ?_0 \;;\; \{U/1,\ X'/1\} \\ \cline{2-2}
\textsc{Failure} & ?_0 \;;\; \{U/1,\ X'/1\} \\ \cline{2-2}
\textsc{Failure} & \varepsilon \;;\; \{U/1,\ X'/1\}
\end{array}$$

**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 $\mathsf{a}$ and the rule $\mathsf{a}\ :-\ \mathsf{a}$. 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).

$$
\begin{array}{rc}
 & \backslash{+}(',' (\mathsf{a}, !))_\varnothing \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Not} & (\mathsf{call}(',' (\mathsf{a}, !)), !_1, \mathsf{fail})_\varnothing \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Call} & (',' (\mathsf{a}, !_2), !_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Conj} & (\mathsf{a}, !_2, !_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Case} & (\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}}_\varnothing \mid (\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}\,:-\,\mathsf{a}}_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Eval} & (!_2, !_1, \mathsf{fail})_\varnothing \mid (\mathsf{a}, !_2, !_1, \mathsf{fail})^{\mathsf{a}\,:-\,\mathsf{a}}_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Cut} & (!_1, \mathsf{fail})_\varnothing \mid ?_2 \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Cut} & \mathsf{fail}_\varnothing \mid ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Fail} & ?_1 \mid ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Failure} & ?_0 \ ; \ \varepsilon \\ \cline{2-2}
\textsc{Failure} & \varepsilon \ ; \ \varepsilon
\end{array}
$$

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

## 4 "All Solution" Predicates

We now consider the unification predicate = and the predicates findall, bagof, 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. App. B.3).

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 $\sigma_1, \dots, \sigma_n$ are the resulting answer substitutions, then finally the list $[r\sigma_1, \dots, r\sigma_n]$ 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

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

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

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

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

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

$$
\begin{array}{l|c}
 & \mathsf{findall}(U, \mathsf{member}(U, [1]), L)_{\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Findall} & \mathsf{call}(\mathsf{member}(U, [1]))_{\varnothing} \mid \%^{U,[\,],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Call} & \mathsf{member}(U, [1])_{\varnothing} \mid ?_1 \mid \%^{U,[\,],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Case} & \mathsf{member}(U, [1])^{(3)'[!/!_2]}_{\varnothing} \mid \mathsf{member}(U, [1])^{(4)'[!/!_2]}_{\varnothing} \mid ?_2 \mid ?_1 \mid \%^{U,[\,],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Eval} & (!_2)_{\{U/1,\, X'/1\}} \mid \mathsf{member}(U, [1])^{(4)'[!/!_2]}_{\varnothing} \mid ?_2 \mid ?_1 \mid \%^{U,[\,],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Cut} & \square_{\{U/1,\, X'/1\}} \mid ?_2 \mid ?_1 \mid \%^{U,[\,],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{FindNext} & ?_2 \mid ?_1 \mid \%^{U,[1],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Failure} & ?_1 \mid \%^{U,[1],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Failure} & \%^{U,[1],L}_{\square,\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{FoundAll} & ([1]{=}L)_{\varnothing} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{UnifySuccess} & \square_{\{L/[1]\}} \mid ?_0 \ ; \ \varepsilon \\ \hline
\textsc{Success} & ?_0 \ ; \ \{L/[1]\} \\ \hline
\textsc{Failure} & \varepsilon \ ; \ \{L/[1]\}
\end{array}
$$

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

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 $\%^{r,\ell,s}_{Q,\delta}$, 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 call$(t)$ has been fully evaluated, the first element of the list of goals is a findall-suspension $\%^{r,\ell,s}_{Q,\delta}$. 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 findall$(U, \mathsf{member}(U, [1]), L)$ 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.

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

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

$$\frac{(\mathsf{retract}(c),Q)_\delta \mid S \,;\, \mathcal{D} \,;\, A}{:\!\not-{}_{Q,\delta}^{c,(c_1,m_1)} \mid \cdots \mid :\!\not-{}_{Q,\delta}^{c,(c_a,m_a)} \mid S \,;\, \mathcal{D} \,;\, A}\ (\textsc{Retract})\ \begin{array}{l}\text{if } \mathit{Slice}_{\mathcal{D}}(c) =\\ ((c_1,m_1),\dots,(c_a,m_a))\end{array}$$

$$\frac{:\!\not-{}_{Q,\delta}^{c,(c',m)} \mid S \,;\, \mathcal{D} \,;\, A}{(Q\sigma)_{\delta\sigma} \mid S \,;\, \mathcal{D} \setminus (c',m) \,;\, A}\ (\textsc{RetSuc})\ \begin{array}{l}\text{if } \sigma =\\ \mathit{mgu}(c,c')\end{array} \qquad \frac{:\!\not-{}_{Q,\delta}^{c,(c',m)} \mid S \,;\, \mathcal{D} \,;\, A}{S \,;\, \mathcal{D} \,;\, A}\ (\textsc{RetFail})\ \begin{array}{l}\text{if}\\ \mathit{mgu}(c,c')\\ = \mathit{fail}\end{array}$$

**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.

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

So first the two clauses $\mathsf{p}(\mathsf{a})$ and $\mathsf{p}(\mathsf{b})$ are asserted, i.e., $\mathcal{D}$ contains $(\mathsf{p}(\mathsf{a}),1)$ and $(\mathsf{p}(\mathsf{b}),2)$. When $\mathsf{retract}(\mathsf{p}(X))$ 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 $\mathit{Slice}_{\mathcal{D}}(c) = ((c_1,m_1),\dots,(c_a,m_a))$, cf. the (Retract) inference rule in Fig. 8. Such a retract marker $:\!\not-{}_{Q,\delta}^{c,(c_i,m_i)}$ 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 $:\!\not-{}_{Q,\varnothing}^{\mathsf{p}(X),(\mathsf{p}(\mathsf{a}),1)}$ and $:\!\not-{}_{Q,\varnothing}^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}$, 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. Appendix B.4.

[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 $\mathsf{p}(X)$ :– true). Moreover, $c$ may not have variables on predication positions.

$$\begin{array}{lc}
 & (\mathsf{assertz}(\mathsf{p}(\mathsf{a})), \mathsf{assertz}(\mathsf{p}(\mathsf{b})), \mathsf{retract}(\mathsf{p}(X)), Q)_\varnothing \mid ?_0 \;;\; \varepsilon \;;\; \varepsilon \\ \cline{2-2}
\textsc{AssZ} & (\mathsf{assertz}(\mathsf{p}(\mathsf{b})), \mathsf{retract}(\mathsf{p}(X)), Q)_\varnothing \mid ?_0 \;;\; (\mathsf{p}(\mathsf{a}), 1) \;;\; \varepsilon \\ \cline{2-2}
\textsc{AssZ} & (\mathsf{retract}(\mathsf{p}(X)), Q)_\varnothing \mid ?_0 \;;\; (\mathsf{p}(\mathsf{a}), 1) \mid (\mathsf{p}(\mathsf{b}), 2) \;;\; \varepsilon \\ \cline{2-2}
\textsc{Retract} & :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{a}),1)}_{Q,\varnothing} \mid :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing} \mid ?_0 \;;\; (\mathsf{p}(\mathsf{a}), 1) \mid (\mathsf{p}(\mathsf{b}), 2) \;;\; \varepsilon \\ \cline{2-2}
\textsc{RetSuc} & (Q[X/\mathsf{a}])_{\{X/\mathsf{a}\}} \mid :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing} \mid ?_0 \;;\; (\mathsf{p}(\mathsf{b}), 2) \;;\; \varepsilon \\
\vdots & \\ \cline{2-2}
\textsc{RetSuc} & (\mathsf{assertz}(\mathsf{p}(\mathsf{b})), \mathsf{fail})_{\{X/\mathsf{a}\}} \mid :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing} \mid ?_0 \;;\; \varepsilon \;;\; \varepsilon \\ \cline{2-2}
\textsc{AssZ} & \mathsf{fail}_{\{X/\mathsf{a}\}} \mid :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing} \mid ?_0 \;;\; (\mathsf{p}(\mathsf{b}), 3) \;;\; \varepsilon \\ \cline{2-2}
\textsc{Fail} & :\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing} \mid ?_0 \;;\; (\mathsf{p}(\mathsf{b}), 3) \;;\; \varepsilon \\ \cline{2-2}
\textsc{RetSuc} & (Q[X/\mathsf{b}])_{\{X/\mathsf{b}\}} \mid ?_0 \;;\; (\mathsf{p}(\mathsf{b}), 3) \;;\; \varepsilon \\
\vdots & \\ \cline{2-2}
\textsc{Failure} & \varepsilon \;;\; (\mathsf{p}(\mathsf{b}), 3) \;;\; \varepsilon
\end{array}$$

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

with p(a), the first clause $(\mathsf{p}(\mathsf{a}), 1)$ 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 $(\mathsf{p}(\mathsf{b}), 2)$ is removed from $\mathcal{D}$ and a new clause $(\mathsf{p}(\mathsf{b}), 3)$ 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 $:\not-^{\mathsf{p}(X),(\mathsf{p}(\mathsf{b}),2)}_{Q,\varnothing}$ succeeds since p($X$) also unifies with the clause $(\mathsf{p}(\mathsf{b}), 2)$. However, this retract-statement does not modify $\mathcal{D}$ anymore, since $(\mathsf{p}(\mathsf{b}), 2)$ 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 $(\mathsf{p}(\mathsf{b}), 3)$. 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

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

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

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

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

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

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

**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 $:\not-$ 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. Appendix B.

$$
\begin{array}{rc}
 & \mathsf{catch}(\mathsf{catch}(\mathsf{findall}(X, \mathsf{p}(X), L), \mathsf{a}, \mathsf{fail}), \mathsf{b}, \mathsf{true})_{\varnothing,\varepsilon} \mid ?_0 \\ \hline
\textsc{Catch} & \mathsf{call}(\mathsf{catch}(\mathsf{findall}(X, \mathsf{p}(X), L), \mathsf{a}, \mathsf{fail})_{\varnothing,\,(1,\mathsf{b},\mathsf{true},\square,\varnothing)} \mid ?_1 \mid ?_0 \\ \hline
\textsc{Call} & \mathsf{catch}(\mathsf{findall}(X, \mathsf{p}(X), L), \mathsf{a}, \mathsf{fail})_{\varnothing,\,(1,\mathsf{b},\mathsf{true},\square,\varnothing)} \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Catch} & \mathsf{call}(\mathsf{findall}(X, \mathsf{p}(X), L))_{\varnothing,C} \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Call} & \mathsf{findall}(X, \mathsf{p}(X), L)_{\varnothing,C} \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Findall} & \mathsf{call}(\mathsf{p}(X))_{\varnothing,C} \mid \%^{X,[\,],L}_{\square,\varnothing,C} \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Call} & \mathsf{p}(X)_{\varnothing,C} \mid \%^{X,[\,],L}_{\square,\varnothing,C} \mid ?_5 \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Case} & \mathsf{p}(X)^{\mathsf{p}(\mathsf{a})}_{\varnothing,C} \mid \mathsf{p}(X)^{\mathsf{p}(Y)\,:\!-\,\mathsf{throw}(\mathsf{b})}_{\varnothing,C} \mid ?_6 \mid \%^{X,[\,],L}_{\square,\varnothing,C} \mid ?_5 \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Eval} & \square_{\{X/\mathsf{a}\},\,C} \mid \mathsf{p}(X)^{\mathsf{p}(Y)\,:\!-\,\mathsf{throw}(\mathsf{b})}_{\varnothing,C} \mid ?_6 \mid \%^{X,[\,],L}_{\square,\varnothing,C} \mid ?_5 \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{FindNext} & \mathsf{p}(X)^{\mathsf{p}(Y)\,:\!-\,\mathsf{throw}(\mathsf{b})}_{\varnothing,C} \mid ?_6 \mid \%^{X,[\mathsf{a}],L}_{\square,\varnothing,C} \mid ?_5 \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{Eval} & \mathsf{throw}(\mathsf{b})_{\{Y/X\},\,C} \mid ?_6 \mid \%^{X,[\mathsf{a}],L}_{\square,\varnothing,C} \mid ?_5 \mid ?_4 \mid ?_3 \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{ThrowNext} & \mathsf{throw}(\mathsf{b})_{\{Y/X\},\,(1,\mathsf{b},\mathsf{true},\square,\varnothing)} \mid ?_2 \mid ?_1 \mid ?_0 \\ \hline
\textsc{ThrowSuccess} & \mathsf{call}(\mathsf{true})_{\{Y/X\},\,\varepsilon} \mid ?_0 \\
\vdots &
\end{array}
$$

**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, 14]. 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 the appendix.

**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 $s_0 \leadsto s_1$ denote that the state $s_0$ was transformed to the state $s_1$ by one of our inference rules. Any finite or infinite sequence $s_0 \leadsto s_1 \leadsto s_2 \leadsto \ldots$ 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 $s_Q \leadsto^* \langle S; \mathcal{D}', A'\rangle$. As usual, $\leadsto^*$ denotes the transitive and reflexive closure of $\leadsto$.*

In contrast to Def. 2, the ISO standard [11, 14] 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 (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 Appendix C 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 p(a) and the query 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 $\ell = 3 \cdot k + 1 = 4$.

$$\begin{array}{rc}
 & (\mathsf{p(b)})_{\varnothing} \mid ?_0 \\ \cline{2-2}
\textsc{Case} & (\mathsf{p(b)})_{\varnothing}^{\mathsf{p(a)}} \mid ?_1 \mid ?_0 \\ \cline{2-2}
\textsc{Backtrack} & ?_1 \mid ?_0 \\ \cline{2-2}
\textsc{Failure} & ?_0 \\ \cline{2-2}
\textsc{Failure} & \varepsilon
\end{array}$$

**Fig. 12.** Evaluation for p(b)

## 8 Conclusion

We have presented a new operational semantics for full Prolog (as defined in the corresponding ISO standard [11, 14]) 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 [20, 21].

In [20, 21], we already successfully used a subset of our new semantics for automated termination analysis of definite logic programs with cuts. In future

[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.).

work, we will extend termination analysis to deal with all our inference rules in 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 10967-1. *Information technology - Language independent arithmetic*. 1994.
14. ISO/IEC 13211-1. *Information technology - Programming languages - Prolog*. 1995.
15. J. Jeavons. An alternative linear semantics for allowed logic programs. *Annals of Pure and Applied Logic*, 84(1):3–16, 1997.
16. N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *SLP '84*, pages 281–288. IEEE Press, 1984.
17. M. Kulaš and C. Beierle. Defining standard Prolog in rewriting logic. In *WRLA '00*, ENTCS 36, 2001.
18. T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11:650–665, 1989.
19. 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.
20. 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.
21. 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.
22. H. Zankl and M. Korp. Modular complexity analysis via relative complexity. In *RTA '10*, LIPIcs 6, pages 385–400, 2010.

# Appendices

Appendix A introduces all further notions and notations needed to define our semantics. Then the set of all inference rules with all features of *full* Prolog is given in Appendix B. These rules are grouped according to the classification of built-in predicates in [11]. In Appendix C, we briefly recapitulate the operational semantics of Prolog according to the ISO standard in [11] and then prove the equivalence between our semantics and the ISO semantics.

## A Further Notions and Notations

To handle full Prolog, we extend our states by two additional components: an *environment* $\mathcal{E}$ and a finite set of user-defined *predicate indicators* $\mathcal{PI}$. Most components of the states are only needed for very specific inference rules. Hence, to increase readability, we now denote a state as $\langle S\ ;\ M\rangle$ where $S$ is a sequence of goals and $M$ is a 4-tuple $(\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)$. Furthermore, we add a special state HALT to our set of states to indicate that a halting predicate (halt/0 or halt/1) was executed.

Moreover, we use some further notations and concepts established in the ISO standard for Prolog. For further details we refer to [11, 14]. We denote the set of integers by $\mathbb{I}$ and the set of floats by $\mathbb{F}$. A *number* is either an integer or a float. A *constant* is either a number or an *atom*, where the latter corresponds to the name of a function symbol (which is not a number). The ISO standard also defines several character sets from which atoms can be built. If only one character forms an atom, it is also called a *one-char atom*. Characters itself cannot be arguments of terms in Prolog. Therefore, one-char atoms are often used for character processing. Additionally, each one-char atom is assigned an integer as its *atom code*. For characters, the same assignment is made (i.e., the character code of a character is the same as the atom code for the one-char atom formed by that character). A *list* is a term $.(t_1, .(t_2, \ldots .(t_n, [\,]) \ldots))$, often written as $[t_1, t_2, \ldots, t_n]$. A *partial list* is a term $.(t_1, .(t_2, \ldots .(t_n, X) \ldots))$ where $X$ is a variable, often written as $[t_1, t_2, \ldots, t_n \mid X]$. A term is *callable* if it is no variable and does not have a number at a predication position. A clause can either be *private* or *public*. All dynamic clauses must be public while the static clauses are private by default. However, they can be defined being public by the directive public/1. As we do not consider the preparation of a Prolog program for execution, but assume to be given an already prepared program, we only use an additional set $\mathcal{P}_{public} \subseteq \mathcal{P}$ to mark those static predicates which are public. As this set remains unchanged during execution, we do not add it to our states, but treat it analogously to the set of static clauses $\mathcal{P}$. The *evaluate* function is used to evaluate arithmetic operations and is defined according to [11, 13, 14]. In particular, it leaves the order of argument evaluations implementation-defined. If the arithmetic evaluation results in an error, the function *evaluate* returns a special error value which is not in $\mathbb{I}$, $\mathbb{F}$, or $\{\mathsf{true}, \mathsf{false}\}$. Thus, the rules for built-in predicates for arithmetic comparisons are only applicable if the arithmetic evaluation does not lead to an error. The *term_precedes* relation is used to specify an order between terms in Prolog and is defined according to [11, 14]. In particular, it leaves the order of variables implementation-defined.

The environment $\mathcal{E}$ contains the operator table, the character conversion table, the Prolog flags, and the input and output streams. We refrain from repeating all details about the environment and the side effects performed for the built-in predicates dealing with it. Instead, we refer to [11, 14] and only mention in our rules for these predicates that they perform the same side effects. However, in order to consider all error conditions for built-in predicates, we sometimes refer to some parts and concepts of the environment. For each stream there is an implementation-dependent *stream-term* associated with the stream. Moreover, there might also be other terms denoting this stream as *alias terms*. We consider both kinds of such terms as "*terms denoting a stream*". A stream has two special positions, namely the *end-of-stream position* and the *past-end-of-stream position*. These positions are used in some conditions for our rules.

A *predicate indicator* is an expression $p/a$ where $p$ is an atom and $a$ is an integer with $0 \leq a \leq \mathit{maxarity}$, the latter being an implementation-defined integer denoting the maximal arity a function symbol may have. The set $\mathcal{PI}$ stores all predicate indicators of user-defined predicates. In the initial state, it contains all predicate indicators of the heads of the program clauses (both dynamic and static). Every time a clause is asserted, the corresponding predicate indicator is added to the set $\mathcal{PI}$. However, retracting does not cause any alterations of this set, since it is possible to have no clauses for a predicate, but Prolog still considers it as a known user-defined predicate. Only abolishing the corresponding predicate indicator removes it from the set $\mathcal{PI}$.

According to the extensions introduced in this appendix, the initial state for a query $Q$ and a program with static clauses $\mathcal{P}$ and dynamic clauses $\overline{\mathcal{D}}$ is now defined as $\langle\, Q[!/!_0] \mid ?_0 \;;\; (\mathcal{D} \;;\; \mathcal{E} \;;\; \{\mathit{root}(h) \mid h :\!- B \in \mathcal{P} \mid \overline{\mathcal{D}}\} \;;\; \varepsilon)\,\rangle$ where $\mathcal{D}$ results from $\overline{\mathcal{D}}$ by labeling all clauses in $\overline{\mathcal{D}}$ with pairwise different fresh natural numbers and $\mathcal{E}$ is the implementation-dependent initial environment.

As an additional notation, we sometimes write $s \sim t$ if $s$ and $t$ are two terms which are unifiable, but we are not interested in a particular unifier.

## B Full Rule Set

### B.1 Definite Logic Programming

$$\frac{(t, Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(t, Q)_{\delta,C}^{c_1[!/!_m]} \mid \cdots \mid (t, Q)_{\delta,C}^{c_a[!/!_m]} \mid ?_m \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)} \ (\textsc{Case})$$

if $\mathit{root}(t) \in \mathcal{PI}$, $\mathit{Slice}_{(\mathcal{P}|\overline{\mathcal{D}})}(t) = (c_1, \ldots, c_a)$ with $a \geq 0$, $\overline{\mathcal{D}}$ is $\mathcal{D}$ without clause labels, and $m$ is fresh

$$\frac{\square_{\delta,\varepsilon} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A \mid \delta)} \;(\textsc{Success})$$

if $S$ does not contain any findall-suspension of the form $\%^{r,\ell,s}_{Q,\delta',C}$

$$\frac{?_m \mid S \;;\; M}{S \;;\; M} \;(\textsc{Failure}) \qquad \frac{(t,Q)^{h\,:\!-\,B}_{\delta,C} \mid S \;;\; M}{S \;;\; M} \;(\textsc{Backtrack}) \text{ if } mgu(t,h) = \mathit{fail}$$

$$\frac{(t,Q)^{h\,:\!-\,B}_{\delta,C} \mid S \;;\; M}{(B\sigma,Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \;(\textsc{Eval}) \text{ if } \sigma = mgu(t,h)$$

$$\frac{(t,Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \;(\textsc{UnknownFailure})$$

if $root(t)$ is neither a built-in predicate nor belongs to the set $\mathcal{PI}$ and the flag unknown has the value failure

$$\frac{(t,Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{UnknownWarning})$$

if $root(t)$ is neither a built-in predicate nor belongs to the set $\mathcal{PI}$, the flag unknown has the value warning, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

## B.2 Logic and Control

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

$$\frac{(\mathsf{catch}(t,c,r),Q)_{\delta,C} \mid S \;;\; M}{\mathsf{call}(t)_{\varnothing,C|(m,c,r,Q,\delta)} \mid ?_m \mid S \;;\; M} \;(\textsc{Catch}) \;\begin{array}{l}\text{if } m \text{ is fresh and}\\ t \text{ is callable}\end{array}$$

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

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

$$\frac{(';'(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{(t_1,Q)_{\delta,C} \mid (t_2,Q)_{\delta,C} \mid S \;;\; M} \;(\textsc{Disj}) \text{ if } root(t_1) \neq \text{->/2}$$

$$\frac{(t, Q)_{\delta,C} \mid S \;;\; M}{(\mathsf{throw}(e), Q)_{\delta,C} \mid S \;;\; M}\ (\textsc{Error})$$

if $t$ satisfies at least one error condition according to [11, 14] and $e$ is the corresponding error term

$$\frac{(\mathsf{fail}, Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M}\ (\textsc{Fail}) \qquad \frac{(\mathsf{halt}, Q)_{\delta,C} \mid S \;;\; M}{\mathsf{HALT}}\ (\textsc{Halt})$$

$$\frac{(\mathsf{halt}(t), Q)_{\delta,C} \mid S \;;\; M}{\mathsf{HALT}}\ (\textsc{Halt1})\ \text{ if } t \in \mathbb{I}$$

$$\frac{('\text{->}'(t_1, t_2), Q)_{\delta,C} \mid S \;;\; M}{(\mathsf{call}(t_1), !_m, t_2, Q)_{\delta,C} \mid ?_m \mid S \;;\; M}\ (\textsc{IfThen})\ \text{ if } m \in \mathbb{N} \text{ is fresh}$$

$$\frac{(';'\,('\text{->}'(t_1, t_2), t_3), Q)_{\delta,C} \mid S \;;\; M}{(\mathsf{call}(t_1), !_m, t_2, Q)_{\delta,C} \mid t_3, Q)_{\delta,C} \mid ?_m \mid S \;;\; M}\ (\textsc{IfThenElse})\ \text{ if } m \in \mathbb{N} \text{ is fresh}$$

$$\frac{(\backslash\!+\!(t), Q)_{\delta,C} \mid S \;;\; M}{(\mathsf{call}(t), !_m, \mathsf{fail})_{\delta,C} \mid Q_{\ \delta,C} \mid ?_m \mid S \;;\; M}\ (\textsc{Not})\ \begin{array}{l}\text{if } t \text{ is callable and}\\ m \in \mathbb{N} \text{ fresh}\end{array}$$

$$\frac{(\mathsf{once}(t), Q)_{\delta,C} \mid S \;;\; M}{(\mathsf{call}(','\,(t, !)), Q)_{\delta,C} \mid S \;;\; M}\ (\textsc{Once})\ \text{ if } t \text{ is callable}$$

$$\frac{(\mathsf{repeat}, Q)_{\delta,C} \mid S \;;\; M}{Q_{\ \delta,C} \mid (\mathsf{repeat}, Q)_{\delta,C} \mid S \;;\; M}\ (\textsc{Repeat})$$

$$\frac{(\mathsf{throw}(e), Q)_{\theta,C|(m,c,r,Q',\delta)} \mid S' \mid ?_m \mid S \;;\; M}{(\mathsf{throw}(e), Q)_{\theta,C} \mid S \;;\; M}\ (\textsc{ThrowNext})$$

if $e \notin \mathcal{V}$ and $mgu(c, e') = \mathit{fail}$ for a fresh variant $e'$ of $e$

$$\frac{(\mathsf{throw}(e), Q)_{\theta,C|(m,c,r,Q',\delta)} \mid S' \mid ?_m \mid S \;;\; M}{(\mathsf{call}(r\sigma), Q'\sigma)_{\delta\sigma,C} \mid S \;;\; M}\ (\textsc{ThrowSuccess})$$

if $e \notin \mathcal{V}$ and $\sigma = mgu(c, e')$ for a fresh variant $e'$ of $e$

$$\frac{(\mathsf{throw}(e), Q)_{\theta,\varepsilon} \mid S \;;\; M}{\mathsf{ERROR}}\ (\textsc{ThrowErr})\ \text{ if } e \notin \mathcal{V}$$

$$\frac{(\mathsf{true}, Q)_{\delta,C} \mid S \;;\; M}{Q_{\ \delta,C} \mid S \;;\; M}\ (\textsc{True})$$

## B.3 All Solutions

$$\frac{(\mathsf{findall}(r,t,s),Q)_{\delta,C} \mid S \;;\; M}{\mathsf{call}(t)_{\varnothing,C} \mid \%^{r,[],s}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{Findall})$$

if $t$ is callable, and $s$ is either a variable, a partial list, or a list

$$\frac{\%^{r,\ell,s}_{Q,\delta,C} \mid S \;;\; M}{(\ell{=}s,Q)_{\delta,C} \mid S \;;\; M}\;(\textsc{FoundAll})$$

$$\frac{\square_{\theta,C} \mid S' \mid \%^{r,\ell,s}_{Q,\delta,C'} \mid S \;;\; M}{S' \mid \%^{r,\ell|r\theta,s}_{Q,\delta,C'} \mid S \;;\; M}\;(\textsc{FindNext})$$

if $S'$ contains no $\mathsf{findall}$-suspensions and ($C$ is either empty or else its last element is $(m,c,r,Q,\delta)$ and $S'$ contains no $?_m$ )

$$\frac{(\mathsf{bagof}(r,b,\ell),Q)_{\delta,C} \mid S \;;\; M}{\mathsf{findall}([f(X_1,\dots,X_n),r],t,Y)_{\varnothing,C} \mid \$^{\ell}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{Bagof})$$

if $b = \hat{}\,(t_1, \hat{}\,(t_2, \hat{}\,(\dots, \hat{}\,(t_m,t)\dots)))$ for some $m \geq 0$ and $root(t) \neq \hat{}/2$, $\mathcal{V}(t) \setminus (\bigcup_{i=1}^{m} \mathcal{V}(t_i)) = \{X_1,\dots,X_n\}$, $Y \in \mathcal{V}$ is fresh, $t$ is callable, and $\ell$ is either a variable, a partial list, or a list

$$\frac{\%^{r,\ell,s}_{\square,\varnothing,C} \mid \$^{\ell'}_{Q,\delta,C} \mid S \;;\; M}{\$^{\ell',s\sigma}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{FoundBag})$$

if $\sigma = mgu(\ell,s)$

$$\frac{\$^{\ell,s}_{Q,\delta,C} \mid S \;;\; M}{(\ell\sigma = \ell', Q\sigma)_{\delta\sigma,C} \mid \$^{\ell,s'}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{NextBag})$$

if $[w,t]$ is some element[16] of $s$ and the sublist $s''$ of $s$ consists of all elements $[w',t']$ of $s$ such that $w$ and $w'$ are variants (the order of these elements in $s''$ is the same as in $s$), the substitution $\sigma$ is the $mgu$ of all $w'$ such that there is some $[w',t']$ in $s''$, the list $\ell'$ consists of all terms $t'\sigma$ such that there is some $[w',t']$ in $s''$ (the order of the elements in $\ell'$ corresponds to the order of the elements in $s''$), and $s'$ is the list resulting from removing all elements in $s''$ from $s$

$$\frac{\$^{\ell,[]}_{Q,\delta,C} \mid S \;;\; M}{S \;;\; M}\;(\textsc{EmptyBag})$$

$$\frac{(\mathsf{setof}(r,b,\ell),Q)_{\delta,C} \mid S \;;\; M}{\mathsf{findall}([f(X_1,\dots,X_n),r],t,Y)_{\varnothing,C} \mid \&^{\ell}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{Setof})$$

if $b = \hat{}\,(t_1, \hat{}\,(t_2, \hat{}\,(\dots, \hat{}\,(t_m,t)\dots)))$ for some $m \geq 0$ and $root(t) \neq \hat{}/2$, $\mathcal{V}(t) \setminus (\bigcup_{i=1}^{m} \mathcal{V}(t_i)) = \{X_1,\dots,X_n\}$, $Y \in \mathcal{V}$ is fresh, $t$ is callable, and $\ell$ is either a variable, a partial list, or a list

[16] The choice is undefined according to [11, 14].

$$\frac{\%^{r,\ell,s}_{\Box,\varnothing,C} \mid \&^{\ell'}_{Q,\delta,C} \mid S \;;\; M}{\&^{\ell',s\sigma}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{FoundSet})\quad \text{if } \sigma = mgu(\ell, s)$$

$$\frac{\&^{\ell,s}_{Q,\delta,C} \mid S \;;\; M}{(\ell\sigma = \ell', Q\sigma)_{\delta\sigma,C} \mid \&^{\ell,s'}_{Q,\delta,C} \mid S \;;\; M}\;(\textsc{NextSet})$$

if $[w, t]$ is some element[17] of $s$ and the sublist $s''$ of $s$ consists of all elements $[w', t']$ of $s$ such that $w$ and $w'$ are variants where duplicates are removed from $s''$ and $s''$ is ordered according to the *term_precedes* order, the substitution $\sigma$ is the *mgu* of all $w'$ such that there is some $[w', t']$ in $s''$, the list $\ell'$ consists of all terms $t'\sigma$ such that there is some $[w', t']$ in $s''$ (the order of the elements in $\ell'$ corresponds to the order of the elements in $s''$), and $s'$ is the list resulting from removing all elements in $s''$ from $s$

$$\frac{\&^{\ell,[\,]}_{Q,\delta,C} \mid S \;;\; M}{S \;;\; M}\;(\textsc{EmptySet})$$

## B.4 Clause Creation and Destruction

$$\frac{(\mathsf{abolish}(p/a), Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D}', \mathcal{E}, \mathcal{PI} \setminus \{p/a\}, A)}\;(\textsc{Abolish})$$

if $p$ is an atom, $a$ is an integer with $0 \leq a \leq \mathit{maxarity}$, $p/a$ is not a static predicate, and $\mathcal{D}'$ results from $\mathcal{D}$ by removing all clauses whose heads have the root symbol $p/a$

$$\frac{(\mathsf{asserta}(c), Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \;;\; ((h :\!-\, B', m) \mid \mathcal{D}, \mathcal{E}, \mathcal{PI} \cup \{\mathit{root}(h)\}, A)}\;(\textsc{AssA})$$

if $((\mathit{root}(c) = :\!-/2$ and $c = h :\!-\, B)$ or $(\mathit{root}(c) \neq :\!-/2$, $h = c$ and $B = \mathsf{true}))$, $m \in \mathbb{N}$ is fresh, $h$ and $B' := B[\mathcal{V}/\mathsf{call}(\mathcal{V})]$ are callable, and $\mathit{root}(h)$ is not a static predicate. Here, we assume that $\mathit{root}(h)$ also returns the arity of $h$'s root symbol.

$$\frac{(\mathsf{assertz}(c), Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D} \mid (h :\!-\, B', m), \mathcal{E}, \mathcal{PI} \cup \{\mathit{root}(h)\}, A)}\;(\textsc{AssZ})$$

if $((\mathit{root}(c) = :\!-/2$ and $c = h :\!-\, B)$ or $(\mathit{root}(c) \neq :\!-/2$, $h = c$ and $B = \mathsf{true}))$, $m \in \mathbb{N}$ is fresh, $h$ and $B' := B[\mathcal{V}/\mathsf{call}(\mathcal{V})]$ are callable, and $\mathit{root}(h)$ is not a static predicate. Here, we assume that $\mathit{root}(h)$ also returns the arity of $h$'s root symbol.

[17] The choice is undefined according to [11, 14].

$$\frac{(\mathsf{retract}(c), Q)_{\delta,C} \mid S\ ;\ M}{:\!\not-\,^{h\,:\text{-}\,B,(c_1,m_1)}_{Q,\delta,C} \mid \cdots \mid :\!\not-\,^{h\,:\text{-}\,B,(c_a,m_a)}_{Q,\delta,C} \mid S\ ;\ M}\ (\textsc{Retract})$$

if $((root(c) = \text{:–}/2$ and $c = h \text{ :– } B)$ or $(root(c) \neq \text{:–}/2$, $h = c$ and $B = \mathsf{true}))$, $h$ is callable, $root(h)$ is not a static predicate, and $Slice_{\mathcal{D}}(h) = ((c_1, m_1), \ldots, (c_a, m_a))$

$$\frac{:\!\not-\,^{c,(c',m)}_{Q,\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D} \setminus (c', m), \mathcal{E}, \mathcal{PI}, A)}\ (\textsc{RetSuc})\ \text{ if } \sigma = mgu(c, c')$$

$$\frac{:\!\not-\,^{c,(c',m)}_{Q,\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{RetFail})\ \text{ if } mgu(c, c') = \mathit{fail}$$

## B.5 Arithmetic Comparison

$$\frac{('\triangleright'(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ M}{Q_{\ \delta,C} \mid S\ ;\ M}\ (\textsc{ArithCompSuc})$$

if $evaluate('\triangleright'(t_1, t_2)) = \mathsf{true}$, $\triangleright \in \{=:=, =\backslash=, >, >=, <, =<\}$.

$$\frac{('\triangleright'(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{ArithCompFail})$$

if $evaluate('\triangleright'(t_1, t_2)) = \mathsf{false}$, $\triangleright \in \{=:=, =\backslash=, >, >=, <, =<\}$.

## B.6 Arithmetic Evaluation

$$\frac{(\mathsf{is}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{IsFail})\ \text{ if } evaluate(t_2) = v \in \mathbb{I} \cup \mathbb{F} \text{ and } mgu(t_1, v) = \mathit{fail}$$

$$\frac{(\mathsf{is}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ M}{Q\sigma_{\ \delta\sigma,C} \mid S\ ;\ M}\ (\textsc{IsSuccess})\ \text{ if } evaluate(t_2) = v \in \mathbb{I} \cup \mathbb{F} \text{ and } mgu(t_1, v) = \sigma$$

## B.7 Atomic Term Processing

$$\frac{(\mathsf{atom\_chars}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ M}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ M}\ (\textsc{AtomCharsSuc})$$

if ($t_1$ is an atom, $\ell$ is the list of one-char atoms identical to the sequence of characters of the name of $t_1$, and $\sigma = mgu(\ell, t_2)$) or ($t_1 \in \mathcal{V}$, $t_2$ is a list of one-char atoms, $a$ is the atom whose name is the sequence of the one-char atoms in $t_2$, and $\sigma = \{t_1/a\}$)

$$\frac{(\mathsf{atom\_chars}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \;(\textsc{AtomCharsFail})$$

if $t_1$ is an atom, $\ell$ is the list of one-char atoms identical to the sequence of characters of the name of $t_1$, and $mgu(\ell,t_2) = \mathit{fail}$

$$\frac{(\mathsf{atom\_codes}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \;(\textsc{AtomCodesSuc})$$

if ($t_1$ is an atom, $\ell$ is the list of atom codes whose corresponding one-char atoms form the name of $t_1$, and $\sigma = mgu(\ell,t_2)$) or ($t_1 \in \mathcal{V}$, $t_2$ is a list of atom codes, $a$ is the atom whose name is the sequence of the one-char atoms whose atom codes are in $t_2$, and $\sigma = \{t_1/a\}$)

$$\frac{(\mathsf{atom\_codes}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \;(\textsc{AtomCodesFail})$$

if $t_1$ is an atom, $\ell$ is the list of atom codes whose corresponding one-char atoms form the name of $t_1$, and $mgu(\ell,t_2) = \mathit{fail}$

$$\frac{(\mathsf{atom\_concat}(t_1,t_2,t_3),Q)_{\delta,C} \mid S \;;\; M}{(s_1 = r,Q)_{\delta,C} \mid \ldots \mid (s_n = r,Q)_{\delta,C} \mid S \;;\; M} \;(\textsc{AtomConcat})$$

if ($t_3$ is an atom, $t_1$ and $t_2$ are variables or atoms, there are $n$ pairs of atoms $(a_i,b_i)$ such that the characters forming $t_3$ are the characters forming $a_i$ followed by the characters forming $b_i$, $r = (t_1,t_2)$, and $s_i = (a_i,b_i)$ for all $i \in \{1,\ldots,n\}$) or ($t_3$ is a variable, $t_1$ and $t_2$ are atoms, $n = 1$, $r = t_3$, and $s_1$ is the atom formed by the characters forming $t_1$ followed by the characters forming $t_2$)

$$\frac{(\mathsf{atom\_length}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \;(\textsc{AtomLengthSuc})$$

if $t_1$ is an atom, $t_2$ is a variable or a non-negative integer, $n$ is the number of characters forming $t_1$, and $\sigma = mgu(n,t_2)$

$$\frac{(\mathsf{atom\_length}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \;(\textsc{AtomLengthFail})$$

if $t_1$ is an atom, $t_2$ is a non-negative integer, $n$ is the number of characters forming $t_1$, and $mgu(n,t_2) = \mathit{fail}$

$$\frac{(\mathsf{char\_code}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; M}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \;(\textsc{CharCodeSuc})$$

if ($t_1$ is a one-char atom, $t_2$ is a variable or an integer corresponding to a character code, $n$ is the character code for $t_1$, and $\sigma = mgu(n,t_2)$) or ($t_1$ is a variable, $t_2$ is an integer corresponding to a character code, $a$ is the character corresponding to $t_2$, and $\sigma = \{t_1/a\}$)

$$\frac{(\mathsf{char\_code}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{CharCodeFail})$$

if $t_1$ is an atom, $t_2$ is an integer corresponding to a character code, $n$ is the character code for $t_1$, and $mgu(n,t_2) = \mathit{fail}$

$$\frac{(\mathsf{number\_chars}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ M}\ (\textsc{NumberCharsSuc})$$

if ($t_2$ is a list of one-char atoms which is parsable as the number $n$, $t_2$ is a variable or a number, and $\sigma = mgu(n,t_2)$) or ($t_2$ is not a list of one-char atoms, $t_1$ is a number, $\ell$ is the list corresponding to the characters being output by $\mathsf{write\_canonical}(t_1)$, and $\sigma = mgu(\ell,t_2)$)

$$\frac{(\mathsf{number\_chars}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{NumberCharsFail})$$

if ($t_2$ is a list of one-char atoms which is parsable as the number $n$, $t_2$ is a number, and $mgu(n,t_2) = \mathit{fail}$) or ($t_2$ is not a list of one-char atoms, $t_1$ is a number, $\ell$ is the list corresponding to the characters being output by $\mathsf{write\_canonical}(t_1)$, and $mgu(\ell,t_2) = \mathit{fail}$)

$$\frac{(\mathsf{number\_codes}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ M}\ (\textsc{NumberCodesSuc})$$

if ($t_2$ is a list of character codes whose corresponding characters are parsable as the number $n$, $t_2$ is a variable or a number, and $\sigma = mgu(n,t_2)$) or ($t_2$ is not a list of character codes, $t_1$ is a number, $\ell$ is the list of character codes corresponding to the characters being output by $\mathsf{write\_canonical}(t_1)$, and $\sigma = mgu(\ell,t_2)$)

$$\frac{(\mathsf{number\_codes}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{NumberCodesFail})$$

if ($t_2$ is a list of character codes whose corresponding characters are parsable as the number $n$, $t_2$ is a number, and $mgu(n,t_2) = \mathit{fail}$) or ($t_2$ is not a list of character codes, $t_1$ is a number, $\ell$ is the list of character codes corresponding to the characters being output by $\mathsf{write\_canonical}(t_1)$, and $mgu(\ell,t_2) = \mathit{fail}$)

$$\frac{(\mathsf{sub\_atom}(t,b,\ell,a,s),Q)_{\delta,C} \mid S\ ;\ M}{\begin{array}{c}((b_1,\ell_1,a_1,s_1) = (b,\ell,a,s),Q)_{\delta,C} \mid \ldots \mid \\ ((b_n,\ell_n,a_n,s_n) = (b,\ell,a,s),Q)_{\delta,C} \mid S\ ;\ M\end{array}}\ (\textsc{SubAtom})$$

if $t$ is an atom, $b,\ell$ and $a$ are non-negative integers or variables, $s$ is an atom or a variable, $(b_1,\ell_1,a_1,s_1),\ldots,(b_n,\ell_n,a_n,s_n)$ is the sorted list (according to the *term_precedes* relation) of all tuples $(b_i,\ell_i,a_i,s_i)$ such that $s_i$ is an atom formed by $\ell_i$ characters and there are atoms $t_i^1$ and $t_i^2$ consisting of $b_i$ and $a_i$ characters with $t$ formed by the characters of $t_i^1$ followed by the characters of $s_i$ and then followed by the characters of $t_i^2$

## B.8 Byte Input/Output

$$\frac{(\mathsf{get\_byte}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{GetByte1Suc})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = \mathit{mgu}(b, t)$

$$\frac{(\mathsf{get\_byte}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{GetByte1Fail})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\mathit{mgu}(b, t) = \mathit{fail}$

$$\frac{(\mathsf{get\_byte}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S(\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{GetByte2Suc})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a variable or a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = \mathit{mgu}(b, t_2)$

$$\frac{(\mathsf{get\_byte}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{GetByte2Fail})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\mathit{mgu}(b, t_2) = \mathit{fail}$

$$\frac{(\mathsf{peek\_byte}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{PeekByte1Suc})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = \mathit{mgu}(b, t)$

$$\frac{(\mathsf{peek\_byte}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)}\ (\textsc{PeekByte1Fail})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\mathit{mgu}(b, t) = \mathit{fail}$

$$\frac{(\mathsf{peek\_byte}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PeekByte2Suc})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a variable or a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(b, t_2)$

$$\frac{(\mathsf{peek\_byte}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PeekByte2Fail})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a byte or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $b$ is the next byte to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $mgu(b, t_2) = \mathit{fail}$

$$\frac{(\mathsf{put\_byte}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PutByte1})$$

if the current output stream has appropriate properties according to [11, 14], $t$ is a byte, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{put\_byte}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PutByte2})$$

if $t_1$ is a term denoting a stream with appropriate properties according to [11, 14], $t_2$ is a byte, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

## B.9 Character Input/Output

$$\frac{(\mathsf{get\_char}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetChar1Suc})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or a one-char atom or the atom $\mathsf{end\_of\_file}$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or $\mathsf{end\_of\_file}$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c, t_2)$

$$\frac{(\mathsf{get\_char}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetChar1Fail})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a one-char atom or the atom $\mathsf{end\_of\_file}$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or $\mathsf{end\_of\_file}$ in case that $s$ is at the end-of-stream position, and $mgu(c, t) = \mathit{fail}$

$$\frac{(\mathsf{get\_char}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetChar2Suc})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to $[11,14]$, $t_2$ is a variable or a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in $[11,14]$, $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c,t_2)$

$$\frac{(\mathsf{get\_char}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetChar2Fail})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to $[11,14]$, $t_2$ is a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in $[11,14]$, $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $mgu(c,t_2) = \mathit{fail}$

$$\frac{(\mathsf{get\_code}(t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetCode1Suc})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in $[11,14]$, $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c,t_2)$

$$\frac{(\mathsf{get\_code}(t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetCode1Fail})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in $[11,14]$, $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $mgu(c,t) = \mathit{fail}$

$$\frac{(\mathsf{get\_code}(t_1,t_2),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{GetCode2Suc})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to $[11,14]$, $t_2$ is a variable or an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in $[11,14]$, $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c,t_2)$

$$\frac{(\mathsf{get\_code}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (GetCode2Fail)}$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $mgu(c, t_2) = \mathit{fail}$

$$\frac{(\mathsf{peek\_char}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (PeekChar1Suc)}$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c, t_2)$

$$\frac{(\mathsf{peek\_char}(t), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (PeekChar1Fail)}$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $mgu(c, t) = \mathit{fail}$

$$\frac{(\mathsf{peek\_char}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (PeekChar2Suc)}$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a variable or a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c, t_2)$

$$\frac{(\mathsf{peek\_char}(t_1, t_2), Q)_{\delta,C} \mid S\ ;\ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S\ ;\ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (PeekChar2Fail)}$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a one-char atom or the atom end_of_file, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is a one-char atom formed by the next character to be input from $s$ or end_of_file in case that $s$ is at the end-of-stream position, and $mgu(c, t_2) = \mathit{fail}$

$$\frac{(\mathsf{peek\_code}(t), Q)_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \ (\textsc{PeekCode1Suc})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is a variable or an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c, t_2)$

$$\frac{(\mathsf{peek\_code}(t), Q)_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S \ ; \ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \ (\textsc{PeekCode1Fail})$$

if $s$ is the current input stream with appropriate properties according to [11, 14], $t$ is an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $mgu(c, t) = \mathit{fail}$

$$\frac{(\mathsf{peek\_code}(t_1, t_2), Q)_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \ (\textsc{PeekCode2Suc})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is a variable or an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $\sigma = mgu(c, t_2)$

$$\frac{(\mathsf{peek\_code}(t_1, t_2), Q)_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S \ ; \ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \ (\textsc{PeekCode2Fail})$$

if $t_1$ is a term denoting a stream $s$ with appropriate properties according to [11, 14], $t_2$ is an integer corresponding to a character code or $-1$, $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14], $c$ is the character code of the next character to be input from $s$ or $-1$ in case that $s$ is at the end-of-stream position, and $mgu(c, t_2) = \mathit{fail}$

$$\frac{(\mathsf{put\_char}(t), Q)_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \ ; \ (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \ (\textsc{PutChar1})$$

if the current output stream has appropriate properties according to [11, 14], $t$ is a one-char atom representing a character, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{put\_char}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PutChar2})$$

if $t_1$ is a term denoting a stream with appropriate properties according to [11, 14], $t_2$ is a one-char atom representing a character, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{put\_code}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PutCode1})$$

if the current output stream has appropriate properties according to [11, 14], $t$ is an integer corresponding to a character code, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{put\_code}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{PutCode2})$$

if $t_1$ is a term denoting a stream with appropriate properties according to [11, 14], $t_2$ is an integer corresponding to a character code, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{nl},Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{Newline})$$

if the current output stream has appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{nl}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{Newline1})$$

if $t$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

### B.10 Clause Retrieval and Information

$$\frac{(\mathsf{clause}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{\begin{array}{c}((t_1,t_2)=(h_1,B_1),Q)_{\delta,C} \mid \ldots \mid \\ ((t_1,t_2)=(h_n,B_n),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)\end{array}}\ (\textsc{Clause})$$

if $t_1$ is a callable term, $t_2$ is a variable or a callable term, and $\mathit{Slice}_{(\mathcal{P}_{public}|\overline{\mathcal{D}})}(t_1) = (h_1 \mathrel{:-} B_1, \ldots, h_n \mathrel{:-} B_n)$

$$\frac{(\mathsf{current\_predicate}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{\begin{array}{c}(t=p_1/a_1,Q)_{\delta,C} \mid \ldots \mid \\ (t=p_n/a_n,Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)\end{array}}\ (\textsc{CurrentPredicate})$$

if $t$ is a variable or $t = p/a$ with $p$ being an atom and $a$ being an integer with $0 \le a \le \mathit{maxarity}$, and $\mathcal{PI} = \{p_1/a_1, \ldots, p_n/a_n\}$[18]

[18] The order of the $p_i/a_i$ is implementation-defined according to [11, 14].

### B.11 Flag Updates

$$\frac{(\mathsf{current\_prolog\_flag}(t_1, t_2), Q)_{\delta,C} \mid S \;;\; M}{\begin{array}{c}((t_1, t_2) = (f_1, v_1), Q)_{\delta,C} \mid \ldots \mid \\ ((t_1, t_2) = (f_n, v_n), Q)_{\delta,C} \mid S \;;\; M\end{array}} \text{ (CurrentFlag)}$$

if $t_1$ is a variable or an atom and the environment contains the flags $f_1, \ldots, f_n$ with associated values $v_1, \ldots, v_n$[19]

$$\frac{(\mathsf{set\_prolog\_flag}(t_1, t_2), Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (SetFlag)}$$

if $t_1$ is an atom denoting a valid modifiable flag according to [11, 14], $t_2$ is an appropriate value for the flag $t_1$ (in particular, it is no variable), and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

### B.12 Stream Selection and Control

$$\frac{(\mathsf{at\_end\_of\_stream}, Q)_{\delta,C} \mid S \;;\; M}{Q_{\delta,C} \mid S \;;\; M} \text{ (EOSSuc)}$$

if the current input stream is at the stream position end_of_stream or past_end_of_stream

$$\frac{(\mathsf{at\_end\_of\_stream}, Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \text{ (EOSFail)}$$

if the current input stream is not at the stream position end_of_stream or past_end_of_stream

$$\frac{(\mathsf{at\_end\_of\_stream}(t), Q)_{\delta,C} \mid S \;;\; M}{Q_{\delta,C} \mid S \;;\; M} \text{ (EOS1Suc)}$$

if $t$ is a term denoting a stream which is at the stream position end_of_stream or past_end_of_stream

$$\frac{(\mathsf{at\_end\_of\_stream}(t), Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \text{ (EOS1Fail)}$$

if $t$ is a term denoting a stream which is not at the stream position end_of_stream or past_end_of_stream

$$\frac{(\mathsf{close}(s), Q)_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \text{ (Close1)}$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

[19] The order of the flags is implementation-dependent according to [11, 14].

$$\frac{(\mathsf{close}(s,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \quad (\textsc{Close2})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate close-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{current\_input}(t),Q)_{\delta,C} \mid S \;;\; M}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \quad (\textsc{CurrentInputSuc})$$

if $t$ is a variable or a term denoting a stream according to [11, 14], $s$ is the stream-term of the current input stream, and $\sigma = mgu(s,t)$

$$\frac{(\mathsf{current\_input}(t),Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \quad (\textsc{CurrentInputFail})$$

if $t$ is a term denoting a stream according to [11, 14], $s$ is the stream-term of the current input stream, and $mgu(s,t) = \mathit{fail}$

$$\frac{(\mathsf{current\_output}(t),Q)_{\delta,C} \mid S \;;\; M}{(Q\sigma)_{\delta\sigma,C} \mid S \;;\; M} \quad (\textsc{CurrentOutputSuc})$$

if $t$ is a variable or a term denoting a stream according to [11, 14], $s$ is the stream-term of the current output stream, and $\sigma = mgu(s,t)$

$$\frac{(\mathsf{current\_output}(t),Q)_{\delta,C} \mid S \;;\; M}{S \;;\; M} \quad (\textsc{CurrentOutputFail})$$

if $t$ is a term denoting a stream according to [11, 14], $s$ is the stream-term of the current output stream, and $mgu(s,t) = \mathit{fail}$

$$\frac{(\mathsf{flush\_output},Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \quad (\textsc{FlushOutput})$$

if the current output stream has appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{flush\_output}(t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \quad (\textsc{FlushOutput1})$$

if $t$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{open}(s,m,X),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\{X/t\})_{\delta\{X/t\},C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \quad (\textsc{Open3})$$

if $s$ is a source or sink with appropriate properties according to [11, 14], $m$ is an atom denoting an appropriate mode, $X$ is a variable, $t$ is the stream-term which is to be associated with the stream for $s$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{open}(s, m, X, [o_1, \ldots, o_n]), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\{X/t\})_{\delta\{X/t\},C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{Open4})$$

if $s$ is a source or sink with appropriate properties according to [11, 14], $m$ is an atom denoting an appropriate mode, $X$ is a variable, $n \geq 0$, $o_i$ is an appropriate stream-option according to [11, 14] for all $i \in \{1, \ldots, n\}$, $t$ is the stream-term which is to be associated with the stream for $s$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{set\_input}(t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{SetInput})$$

if $t$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{set\_output}(t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{SetOutput})$$

if $t$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{set\_stream\_position}(t_1, t_2), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{SetStreamPos})$$

if $t_1$ is a term denoting a stream with appropriate properties according to [11, 14], $t_2$ is an appropriate stream position according to [11, 14], and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{stream\_property}(t_1, t_2), Q)_{\delta,C} \mid S \; ; \; M}{\begin{array}{c}((t_1, t_2) = (s_1, p_1), Q)_{\delta,C} \mid \ldots \mid \\ ((t_1, t_2) = (s_n, p_n), Q)_{\delta,C} \mid S \; ; \; M\end{array}} \; (\textsc{StreamProperty})$$

if $t_1$ is a stream-term according to [11, 14], $t_2$ is a stream property according to [11, 14], and $\{(s_1, p_1), \ldots, (s_n, p_n)\}$ is the set of all pairs such that $s_i$ is a currently open stream which has the property $p_i$ for all $i \in \{1, \ldots, n\}$[20]

## B.13 Term Comparison

$$\frac{('\triangleright'(t_1, t_2), Q)_{\alpha,C} \mid S \; ; \; M}{Q_{\;\alpha,C} \mid S \; ; \; M} \; (\textsc{TermCompSuc})$$

and if $\triangleright \in \{@>, @>=, ==, @<, @=<, \backslash==\}$ and the condition *Cond* holds according to Table 1.

$$\frac{('\triangleright'(t_1, t_2), Q)_{\alpha,C} \mid S \; ; \; M}{S \; ; \; M} \; (\textsc{TermCompFail})$$

and if $\triangleright \in \{@>, @>=, ==, @<, @=<, \backslash==\}$ and the condition *Cond* does not hold according to Table 1.

[20] The order of the pairs $(s_i, p_i)$ is implementation-dependent according to [11, 14].

| $\triangleright$ | *Cond* |
|---|---|
| @> | $term\_precedes(t_2, t_1)$ |
| @>= | $term\_precedes(t_2, t_1) \vee t_1 = t_2$ |
| == | $t_1 = t_2$ |
| @< | $term\_precedes(t_1, t_2)$ |
| @=< | $term\_precedes(t_1, t_2) \vee t_1 = t_2$ |
| \== | $t_1 \neq t_2$ |

**Table 1.** Term Comparison Predicates ($' \triangleright' \ (t_1, t_2)$)

## B.14 Term Creation and Decomposition

$$\frac{(\mathsf{arg}(n, f(t_1, \dots, t_k), t), Q)_{\delta,C} \mid S \ ; \ M}{(Q\sigma)_{\delta\sigma,C} \mid S \ ; \ M} \ (\textsc{ArgSuc})$$
if $n$ is an integer with $0 < n \leq k$ and $\sigma = mgu(t_n, t)$

$$\frac{(\mathsf{arg}(n, f(t_1, \dots, t_k), t), Q)_{\delta,C} \mid S \ ; \ M}{S \ ; \ M} \ (\textsc{ArgFail})$$
if $n$ is an integer with $(0 < n \leq k$ and $mgu(t_n, t) = fail)$ or $n = 0$ or $n > k$

$$\frac{(\mathsf{copy\_term}(t_1, t_2), Q)_{\delta,C} \mid S \ ; \ M}{(Q\sigma)_{\delta\sigma,C} \mid S \ ; \ M} \ (\textsc{CopyTermSuc})$$
if $t'_1$ is a fresh variant of $t_1$ and $\sigma = mgu(t'_1, t_2)$

$$\frac{(\mathsf{copy\_term}(t_1, t_2), Q)_{\delta,C} \mid S \ ; \ M}{S \ ; \ M} \ (\textsc{CopyTermFail})$$
if $t'_1$ is a fresh variant of $t_1$ and $mgu(t'_1, t_2) = fail$

$$\frac{(\mathsf{functor}(t, p, n), Q)_{\delta,C} \mid S \ ; \ M}{(Q\sigma)_{\delta\sigma,C} \mid S \ ; \ M} \ (\textsc{FunctorSuc})$$

if ($t = f(t_1, \dots, t_k)$ with $k \geq 0$ and $\sigma = mgu(p/n, f/k)$) or ($t$ is a variable, $p$ is a constant, $n$ is an integer with $0 \leq n \leq maxarity$, $n = 0$ if $p$ is a number, and $\sigma = \{t/p(X_1, \dots, X_n)\}$ where $X_1, \dots, X_n$ are pairwise different fresh variables)

$$\frac{(\mathsf{functor}(f(t_1, \dots, t_k), p, n), Q)_{\delta,C} \mid S \ ; \ M}{S \ ; \ M} \ (\textsc{FunctorFail})$$
if $k \geq 0$ and $mgu(p/n, f/k) = fail$

$$\frac{('{=..}'(t,\ell),Q)_{\delta,C} \mid S\ ;\ M}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ M}\ (\textsc{UnivSuc})$$

if ($t = f(t_1,\dots,t_k)$ with $k \geq 0$, $\ell$ is a variable, partial list, or a list whose first element is a variable or a constant (where this first element must be an atom if the list has a length greater than 1), and $\sigma = mgu([f,t_1,\dots,t_k],\ell)$) or ($t$ is a variable, $\ell = [f,a_1,\dots,a_n]$ with $f$ being an atomic term (an atom if $n > 0$), $0 \leq n \leq maxarity$, and $\sigma = \{t/f(a_1,\dots,a_n)\}$)

$$\frac{({=..}(f(t_1,\dots,t_k),\ell),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{UnivFail})$$

if $k \geq 0$, $\ell$ is a partial list or a list whose first element is a variable or an atomic term (an atom if the list has a length greater than 1), and $mgu([f,t_1,\dots,t_k],\ell) = \mathit{fail}$

### B.15 Term Unification

$$\frac{('\backslash{=}'(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{Q_{\delta,C} \mid S\ ;\ M}\ (\textsc{NoUnifySuccess})\ \text{ if } mgu(t_1,t_2) = \mathit{fail}$$

$$\frac{('\backslash{=}'(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{NoUnifyFail})\ \text{ if } t_1 \sim t_2$$

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

$$\frac{('{=}'(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{UnifyFail})\ \text{ if } mgu(t_1,t_2) = \mathit{fail}$$

$$\frac{(\mathsf{unify\_with\_occurs\_check}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ M}\ (\textsc{UnifyOccurSuc})\ \text{ if } \sigma = mgu(t_1,t_2)$$

$$\frac{(\mathsf{unify\_with\_occurs\_check}(t_1,t_2),Q)_{\delta,C} \mid S\ ;\ M}{S\ ;\ M}\ (\textsc{UnifyOccurFail})\ \text{ if } mgu(t_1,t_2) = \mathit{fail}$$

[21]

[21] The ISO standard [11, 14] does not define unification in cases where it makes a difference whether or not the occurs check is performed. Hence, unify_with_occurs_check/2 behaves identical to =/2. In case one adapts our semantics to use unification without occurs check in general, the predicate unify_with_occurs_check/2 then still performs unification with occurs check. However, in the presence of rational terms which may be constructed due to unification without occurs check, one has also to define how such terms are handled by this predicate. SWI-Prolog does not terminate when it calls this predicate on two identical infinite rational terms, for example.

## B.16 Type Testing

| ttpred | *Cond* |
|---|---|
| atom | $t$ is an atom |
| atomic | $t$ is a constant |
| compound | $t$ is no constant and no variable |
| float | $t \in \mathbb{F}$ |
| integer | $t \in \mathbb{I}$ |
| nonvar | $t \notin \mathcal{V}$ |
| number | $t \in \mathbb{I} \cup \mathbb{F}$ |
| var | $t \in \mathcal{V}$ |

**Table 2.** Type Testing Predicates ($\mathsf{ttpred}(t)$)

$$\frac{(\mathsf{ttpred}(t), Q)_{\delta,C} \mid S \; ; \; M}{Q_{\delta,C} \mid S \; ; \; M} \quad (\textsc{TypeTestSuc})$$

if $\mathsf{ttpred} \in \{\mathsf{atom}, \mathsf{atomic}, \mathsf{compound}, \mathsf{float}, \mathsf{integer}, \mathsf{nonvar}, \mathsf{number}, \mathsf{var}\}$ and the condition *Cond* holds according to Table 2.

$$\frac{(\mathsf{ttpred}(t), Q)_{\delta,C} \mid S \; ; \; M}{S \; ; \; M} \quad (\textsc{TypeTestFail})$$

if $\mathsf{ttpred} \in \{\mathsf{atom}, \mathsf{atomic}, \mathsf{compound}, \mathsf{float}, \mathsf{integer}, \mathsf{nonvar}, \mathsf{number}, \mathsf{var}\}$ and the condition *Cond* does not hold according to Table 2.

## B.17 Term Input/Output

$$\frac{(\mathsf{char\_conversion}(t_1, t_2), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \quad (\textsc{CharConv})$$

if $t_1$ and $t_2$ are one-char atoms and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{current\_char\_conversion}(t_1, t_2), Q)_{\delta,C} \mid S \; ; \; M}{\begin{array}{c}((t_1, t_2) = (i_1, o_1), Q)_{\delta,C} \mid \ldots \mid \\ ((t_1, t_2) = (i_n, o_n), Q)_{\delta,C} \mid S \; ; \; M\end{array}} \quad (\textsc{CurrentCharConv})$$

if $t_1$ and $t_2$ are variables or one-char atoms and $\{(i_1, o_1), \ldots, (i_n, o_n)\}$ is the set of all current character conversions in the character conversion table according to [11, 14][22]

$$\frac{(\mathsf{current\_op}(m, s, t), Q)_{\delta,C} \mid S \; ; \; M}{\begin{array}{c}((m, s, t) = (m_1, s_1, t_1), Q)_{\delta,C} \mid \ldots \mid \\ ((m, s, t) = (m_n, s_n, t_n), Q)_{\delta,C} \mid S \; ; \; M\end{array}} \; (\textsc{CurrentOp})$$

if $m$ is an integer between 0 and 1200 (inclusive), $s$ is an atom denoting a valid operator specifier according to [11, 14], $t$ is an atom, and $\{(m_1, s_1, t_1), \ldots, (m_n, s_n, t_n)\}$ is the set of all current operator entries in the operator table

$$\frac{(\mathsf{op}(m, s, t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{Q_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{Op})$$

if $m$ is an integer between 0 and 1200 (inclusive), $s$ is an atom denoting a valid operator specifier according to [11, 14], $t$ is an atom or a list of atoms not being or containing the atom $','$ and not leading to an invalid operator table, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read}(t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{Read1Suc})$$

if the current input stream $s$ has appropriate properties according to [11, 14], $r$ is the read-term being input from $s$, $\sigma = mgu(t, r)$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read}(t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{Read1Fail})$$

if the current input stream $s$ has appropriate properties according to [11, 14], $r$ is the read-term being input from $s$, $mgu(t, r) = \mathit{fail}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read}(s, t), Q)_{\delta,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}, \mathcal{PI}, A)}{(Q\sigma)_{\delta\sigma,C} \mid S \; ; \; (\mathcal{D}, \mathcal{E}', \mathcal{PI}, A)} \; (\textsc{Read2Suc})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $r$ is the read-term being input from $s$, $\sigma = mgu(t, r)$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

[22] The order of the pairs $(i_j, o_j)$ is implementation-dependent according to [11, 14].

$$\frac{(\mathsf{read}(s,t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{Read2Fail})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $r$ is the read-term being input from $s$, $mgu(t,r) = \mathit{fail}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read\_term}(t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{ReadTerm2Suc})$$

if the current input stream $s$ has appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate read-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, $r$ is the read-term being input from $s$, $\theta = mgu(t,r)$, $\mu$ is the substitution instantiating the read-options $o_1,\ldots,o_n$ according to [11, 14], $\sigma = \theta\mu$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read\_term}(t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{ReadTerm2Fail})$$

if the current input stream $s$ has appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate read-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, $r$ is the read-term being input from $s$, $mgu(t,r) = \mathit{fail}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read\_term}(s,t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{(Q\sigma)_{\delta\sigma,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{ReadTerm3Suc})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate read-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, $r$ is the read-term being input from $s$, $\theta = mgu(t,r)$, $\mu$ is the substitution instantiating the read-options $o_1,\ldots,o_n$ according to [11, 14], $\sigma = \theta\mu$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{read\_term}(s,t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{ReadTerm3Fail})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate read-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, $r$ is the read-term being input from $s$, $mgu(t,r) = \mathit{fail}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write}(t),Q)_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S\ ;\ (\mathcal{D},\mathcal{E}',\mathcal{PI},A)}\ (\textsc{Write1})$$

if the current output stream $s$ has appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write}(s,t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{Write2})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write\_canonical}(t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{WriteCanonical1})$$

if the current output stream $s$ has appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write\_canonical}(s,t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{WriteCanonical2})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write\_term}(t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{WriteTerm2})$$

if the current output stream $s$ has appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate write-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{write\_term}(s,t,[o_1,\ldots,o_n]),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{WriteTerm3})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14], $n \geq 0$, $o_i$ is an appropriate write-option according to [11, 14] for all $i \in \{1,\ldots,n\}$, and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{writeq}(t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{Writeq1})$$

if the current output stream $s$ has appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

$$\frac{(\mathsf{writeq}(s,t),Q)_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E},\mathcal{PI},A)}{Q_{\delta,C} \mid S \;;\; (\mathcal{D},\mathcal{E}',\mathcal{PI},A)} \;(\textsc{Writeq2})$$

if $s$ is a term denoting a stream with appropriate properties according to [11, 14] and $\mathcal{E}'$ results from $\mathcal{E}$ according to the side effects described in [11, 14]

## C Equivalence to the ISO Semantics

We now prove the theorems of Sect. 7 in the main part of the paper. For the definition of our semantics to be deterministic, we need Thm. 1.

**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.*

*Proof.* For each pair of different inference rules one of the following two conditions holds:

– The first goal in the states for which the rules are applicable must have different shapes.
– The conditions of the two rules exclude each other.

Note that the first condition is also true for the (Case) rule compared to any rule for built-in predicates as it must not have a built-in predicate at the "first" predication position in the first goal.

Note that the result of applying an inference rule is always uniquely determined. □

The ISO standard for Prolog [11, 14] defines the *operational semantics* of Prolog programs in terms of Prolog search trees. These trees represent the computation according to the standard. Therefore, we first recapitulate the definition of Prolog search trees as given in [11] using the notations established in the current paper.

**Definition 4 (Prolog Search Tree [11]).** *A* Prolog search tree *is a tree whose nodes have two labels. The first label is a query from $\mathcal{T}(\Sigma, \mathcal{V})^*$ and the second label is a substitution (called the* local substitution*). Additionally, the tree has a set of* unvisited nodes *which is a subset of the nodes in the tree. If the tree is* finished*, it has no* current node *and no* unvisited nodes*. Otherwise it has exactly one current node which belongs to the tree, but not to the set of unvisited nodes.*

The ISO standard describes the operational semantics of Prolog by an algorithm combining the construction and the traversal of Prolog search trees during the execution of a Prolog program w.r.t. a query. Such a Prolog search tree also incorporates the execution of built-in predicates. We recapitulate this algorithm in the following definition.

**Definition 5 (Search Tree Construction and Traversal Algorithm [11]).** *Given a Prolog program $\mathcal{P} \mid \overline{\mathcal{D}}$ and a query $Q$, the* search tree construction and traversal algorithm *works as follows:*

1. *Start from the root as current node, labeled by the initial query $Q$ and by the empty substitution as local substitution.*

2. *If the query $Q$ of the current node is* true *then backtrack to the first unvisited node $n$ w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search tree and continue with Step 2 where the current node is $n$ and $n$ is dropped from the set of unvisited nodes. If there are no unvisited nodes, the execution of the initial query is finished.*
3. *Otherwise let $t$ be the first term in $Q$.*
4. *If $t$ is* true *delete it, and proceed to Step 2 with the new current query being the tail of the sequence $Q$.*
5. *If $t$ corresponds to a user-defined predicate which exists in the data base:*
   (a) *If no renamed clause in $\mathcal{P} \mid \overline{\mathcal{D}}$ has a head which unifies with $t$ then backtrack to the first unvisited node $n$ w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search tree and continue with Step 2 where the current node is $n$ and $n$ is dropped from the set of unvisited nodes.*
   (b) *Otherwise add to the current node as many children as there are freshly renamed clauses $H :- B \in \mathcal{P} \mid \overline{\mathcal{D}}$ whose head is unifiable with $t$. The order of the children corresponds to the order of the clauses in $\mathcal{P} \mid \overline{\mathcal{D}}$. The child nodes are labeled with a local substitution $\sigma = mgu(t, H)$ ($H :- B$ being the corresponding freshly renamed clause), and the query $Q'$ which is $Q\sigma$ in which $t$ has been previously replaced by $B$. The current node becomes the first child and we proceed to Step 2.*
6. *Otherwise, if $t$ corresponds to a built-in predicate: The specific side effects for the built-in predicate described in [11] are performed and the execution continues at Step 2 with or without preceding backtracking or generates an error according to the description of the built-in predicate in [11].*
7. *Otherwise $t$ does not correspond to any existing predicate and the action depends on the value of the flag* unknown.
   – *If this value is* error *then an error is generated whose effect corresponds to executing the built-in predicate* throw(existence_error(procedure, $PI$)) *at the same node where $PI$ is the predicate indicator of $t$.*
   – *If this value is* warning *then an implementation-dependent warning is generated and the current query fails, i.e., we backtrack to the first unvisited node $n$ w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search tree and continue with Step 2 where the current node is $n$ and $n$ is dropped from the set of unvisited nodes. If there are no unvisited nodes, then the execution of the initial query is finished.*
   – *If this value is* failure *then the current query fails, i.e., we backtrack to the first unvisited node $n$ w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search tree and continue with Step 2 where the current node is $n$ and $n$ is dropped from the set of unvisited nodes. If there are no unvisited nodes, the execution of the initial query is finished.*

What happens after the execution of the initial query is finished, is implementation-defined. However, the standard does define answer substitutions for Prolog search trees.

**Definition 6 (Answer Substitution [11]).** *Given a Prolog search tree, an answer substitution for the initial query of that tree is the composition of all local substitutions along a path from the root node to a success node (i.e., a node which is labeled with* true *and some local substitution). The list of answer substitutions computed by the execution of a query is obtained by gathering the answer substitutions in the Prolog search tree in a depth-first, left-to-right search.*

**Definition 7 (Complexity of the Search Tree Construction and Traversal Algorithm).** *For the complexity of the search tree construction and traversal algorithm, we count the number of necessary unification tests where the execution of a built-in predicate without any unification test counts as one unification test. A necessary unification test occurs whenever we test whether two terms unify, unless the corresponding node in the constructed Prolog search tree will be deleted due to a cut, a halting predicate, or an error being thrown (assuming that we would also add nodes for the failing unifications).*

We now prove the central theorem of this paper.

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

*(a) Let $\ell$ be the* length *of $Q$'s derivation 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 $X\theta_i = X\delta_i\,\tau_i$.*

*Proof.* We show the theorem by the following two propositions: First, if the execution of $Q$ is finished after $k$ unification tests (i.e., the execution is terminating), then the following holds.

(a) For the *length* $\ell$ of $Q$'s maximal derivation according to our semantics in Def. 2, we have $k + 1 \leq \ell \leq 3 \cdot k + 1$ if the execution does not lead to a program error or the evaluation of a halting predicate (halt/0 or halt/1). Otherwise, we have $k \leq \ell \leq 3 \cdot k + 1$.
(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 standard.
(c) $Q$ leads to a finite list of *answer substitutions* $[\delta_1, \ldots, \delta_n]$ according to our semantics in Def. 2 iff $Q$ leads to a list of *answer substitutions* $[\theta_1, \ldots, \theta_n]$ according to the ISO standard, where the two lists have the same length $n \in \mathbb{N}$ and for each $1 \leq i \leq n$, there exists a variable renaming $\tau_i$ such that $X\theta_i = X\delta_i\,\tau_i$ for all variables $X \in \mathcal{V}(Q)$.

(d) If the execution does not lead to a program error or the evaluation of a halting predicate, then after the execution of $Q$, the obtained sets of dynamic clauses in our semantics and in the ISO semantics are the same up to variable renaming.

(e) Both the ISO semantics and our semantics cause the same side effects w.r.t. the environment for any query $Q$.

Second, if the execution of $Q$ does not terminate, then we have that there is a finite prefix of the ISO execution obtaining the answer substitutions $\theta_1, \ldots, \theta_n$ iff there is a corresponding derivation reaching a state $\langle S \ ; \ (\mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \delta_1 \mid \ldots \mid \delta_n)\rangle$ such that there are variable renamings $\tau_1, \ldots, \tau_n$ with $X\theta_i = X\delta_i\tau_i$ for all $i \in \{1, \ldots, n\}$ and variables $X \in \mathcal{V}(Q)$.
Clearly, the theorem is implied by these two propositions.

We show the first proposition by induction over the number $k$ of unification tests w.r.t. the operational semantics of the ISO standard. Part (e) of the first proposition is trivially true since we use the same environment as in the ISO semantics and did not modify any operations on it.

If $k = 1$, the algorithm is finished after the first unification test. If the initial goal is true, then the Prolog search tree just consists of the root node with the empty local substitution. The corresponding derivation consists of one application of the (True) rule followed by one application of the (Success) rule and it finishes by one application of the (Failure) rule leading to the empty list of goals with the empty substitution as the only answer substitution. The derivation has, thus, the length $\ell = 3$. Hence, the proposition holds in this case. Otherwise the initial goal has the form $(t, Q)$ where $root(t)$ is no user-defined predicate (otherwise we would have at least one more unification step). If $root(t)$ is the built-in predicate throw/1, then the execution leads to a program error. The derivation consists of one application of the (ThrowErr) rule and, thus, also results in an error and has the length $\ell = 1$. Hence, the proposition holds in this case. If $root(t)$ is the built-in predicate halt/0 or halt/1, then the execution is finished directly after its execution. The corresponding derivation consists of one application of the (Halt) or (Halt1) rule, respectively, and is also finished directly. Since we have $k = \ell = 1$ for the length $\ell$ of the derivation, the proposition holds in this case. If $root(t)$ is the built-in predicate atom_concat/3, then we have $t = \mathsf{atom\_concat}(t_1, t_2, '')$ with $t_1$ and $t_2$ being variables or atoms and $mgu(\,(t_1, t_2)\,,\,('','')\,) = \mathit{fail}$. Here, $''$ is the *null*-atom in Prolog. The corresponding derivation consists of one application of the (AtomConcat) rule followed by one application of the (UnifyFail) rule and finally one application of the (Failure) rule. Thus, the derivation has the length $\ell = 3$ and the proposition holds in this case. If $root(t)$ is another built-in predicate, then its evaluation must fail (otherwise we would perform at least one more unification test in both success and error case for each built-in predicate). Then the execution is finished as there are no unvisited nodes left. The corresponding derivation consists of one application of the particular failing inference rule for the respective built-in predicate followed by one application of the (Failure) rule. The list of answer substitutions is empty in both the ISO execution and the derivation and we have

$\ell = 2$ for the length of the derivation. Thus, the proposition holds in this case. Finally, if $root(t)$ is no built-in predicate, then there is at most one clause in the program whose head has the same root symbol as $t$. If there is such a clause with head $h$, then we must have $mgu(t, h) = fail$ (otherwise we would have more than one unification test). Then the execution is finished as there are no unvisited nodes left. The corresponding derivation consists of one application of the (Case) rule followed by one application of the (Backtrack) rule and finally two applications of the (Failure) rule leading to the empty list of goals with the empty list of answer substitutions. Since the length of the derivation is $\ell = 4$, the proposition holds in this case. If $root(t)$ does not belong to the set $\mathcal{PI}$ and it is no built-in predicate, then the flag unknown must be set to either warning or failure, because otherwise we would have one more unification test. The Prolog search tree just consists of the root node with the empty local substitution, while the corresponding derivation consists of one application of the (UnknownWarning) or (UnknownFailure) rule followed by one application of the (Failure) rule leading to the empty list of goals with no answer substitutions. The length $\ell$ of the derivation is again $\ell = 2$. Hence, the proposition holds in this case. If $root(t) \in \mathcal{PI}$, but $Slice_{(\mathcal{P}|\overline{\mathcal{D}})}(t)$ is empty, then the execution fails. The corresponding derivation consists of one application of the (Case) rule followed by two applications of the (Failure) rule. The length of the derivation is $\ell = 3$ and, thus, the proposition holds in this case.

If $k > 1$, we can assume that the proposition holds for all executions with $k' < k$ unification tests. The initial goal has the form $(t, Q)$ for a term $t$ and a (possibly empty) sequence of terms $Q$. We perform a case analysis over the shape of $t$.

– If $root(t)$ neither belongs to the set $\mathcal{PI}$ nor to the built-in predicates, then the flag unknown must be set to error, because $k > 1$. The execution continues by evaluating the throw/1 predicate leading to a program error, while the corresponding derivation consists of one application of the (Error) rule followed by one application of the (ThrowErr) rule also leading to an error. Since we have $k = \ell = 2$ for the length $\ell$ of the derivation, the proposition holds.
– If $root(t)$ is a user-defined predicate, then we have $Slice_{(\mathcal{P}|\overline{\mathcal{D}})}(t) = (c_1, \ldots, c_n)$ with $n > 0$. Thus, the first $j \leq n$ unification tests are the necessary unification tests for the clauses in $Slice_{(\mathcal{P}|\overline{\mathcal{D}})}(t)$ where $j \in \{1, \ldots, n\}$ is the smallest index such that $c_j$ introduces a cut which will be reached during the remaining execution, or the execution of $(B\sigma, Q\sigma)$ where $c_j = h :- B$ and $mgu(t, h) = \sigma$ leads to an error which is not caught, ends with the execution of a halting predicate, or evaluates a cut from the initial query. If no such $j$ exists, we have $j = n$. The corresponding derivation starts with an application of the (Case) rule resulting in the state $\langle (t', Q')^{c'_1}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ where $t' = t[!/!_0]$, $Q' = Q[!/!_0]$, and $c'_a = c_a[!/!_m]$ for all $a \in \{1, \ldots, n\}$. Let $i \in \{1, \ldots, n\}$ be the smallest index such that the (Eval) rule is applicable to the state $\langle (t', Q')^{c'_i}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$. If no such $i$ exists then we must have $j = n$ and all $n$

unification tests for $t$ and some clause heads in $\mathcal{P} \mid \overline{\mathcal{D}}$ fail. Then the execution is finished since there is no unvisited node left. The corresponding derivation continues with $n$ applications of the (BACKTRACK) rule followed by two applications of the (FAILURE) rule resulting in the state $\langle \varepsilon \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ such that the derivation is finished, too. As we have $k = n$ unification tests and a derivation of length $\ell = n + 3$, we obtain

$$k + 1 = n + 1 < n + 3 = \ell \overset{n>0}{\leq} 3 \cdot n + 1 = 3 \cdot k + 1.$$

Thus, the proposition holds in this case. If an index $i$ as described above exists, we have $i \leq j$, because a cut, a halting predicate, or an error can only be reached if the corresponding unification test succeeds. The execution performs $i-1$ failing unification tests before executing the first successful one and creates a new child for the current node labeled with $(B\sigma, Q\sigma)$ and the local substitution $\sigma$ where $c_i = h \text{ :- } B$ and $mgu(t, h) = \sigma$. The corresponding derivation continues with $i-1$ applications of the (BACKTRACK) rule followed by one application of the (EVAL) rule resulting in the state $\langle (B'\sigma, Q'\sigma)_{\sigma,\varepsilon} \mid (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ where $c'_i = h' \text{ :- } B'$ (as the $mgu$ is unique modulo variable renaming, we can w.l.o.g. assume that both execution and derivation use the same $mgu$). Now consider an execution for the goal $(B\sigma, Q\sigma)$. Since this goal is part of the current execution (where at least one unification test has been performed before), it must have $k'$ unification tests with $k' < k$ and, hence, we can use the induction hypothesis to obtain a derivation of length $\ell'$ simulating the execution. Moreover, we have $\ell' \leq 3 \cdot k' + 1$ and $k' + 1 \leq \ell'$ if the execution of $(B\sigma, Q\sigma)$ does not result in a program error or terminates by a halting predicate. Otherwise we still have $k' \leq \ell'$. We perform a case analysis over the side effects occurring during the execution of the subgoal.

- If the ISO execution reaches a cut which was already at a predication position in the goal $(B\sigma, Q\sigma)$, but does not result in an error or evaluates a halting predicate, so does the derivation. Then we have $i = j$. From the derivation obtained by the induction hypothesis (i.e., $\langle (B\sigma, Q\sigma)[!/!_0]_{\varnothing,\varepsilon} \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle \leadsto^{\ell'} \langle \varepsilon \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A \rangle$) we construct a derivation from the state $\langle (B'\sigma, Q'\sigma)_{\sigma,\varepsilon} \mid (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ to the state $\langle ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A' \rangle$ where $\mathcal{D}'$ is the dynamic part of the program in the last state of the obtained derivation, $\mathcal{E}'$ is the environment in that state, $\mathcal{PI}'$ is the corresponding set of user-defined predicate indicators, and $A'$ is constructed from the list of answer substitutions $A$ in that last state by replacing every $\delta \in A$ by $\sigma\delta$. This new derivation is constructed by first replacing all answer substitutions and candidate answer substitutions $\delta$ in the derivation by $\sigma\delta$. Then we replace all those $!_0$ by $!_m$ which have been at some position in $(B\sigma, Q\sigma)$ where there is a $!_m$ at the same position in $(B'\sigma, Q'\sigma)$. W.l.o.g. we assume that the scope $m$ is not used within the derivation obtained by the induction hypothesis. Then we add the goals $(t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid$

$(t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m$ before the initial scope marker $?_0$ in every state up to the first state where a cut with the initial scope $!_0$ or with the scope $m$ is executed (inclusive). If the evaluated cut has the scope $m$, we add the goal $?_m$ before the initial scope marker to all other states up to the first one where a cut with the initial scope is executed (to all other states if no such state exists). Note that these changes do not modify the length of the derivation. If a cut with the initial scope was evaluated, we drop the last application of the (Failure) rule from the obtained derivation. Now we have reached the state $\langle ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A' \rangle$. Note that the execution can be finished after $k = i + k'$ unification tests as all remaining unvisited nodes have been deleted due to the cut. Now the constructed derivation is finished by applying the (Failure) rule and has, thus, a length of $\ell = i + \ell' + 1$ in case a cut with the initial scope was evaluated. As we have $i > 0$, we obtain

$$k + 1 = i + k' + 1 \leq i + \ell' < i + \ell' + 1 = \ell$$

and

$$\ell = i + \ell' + 1 \leq i + 3 \cdot k' + 2 \overset{i>0}{<} 3 \cdot i + 3 \cdot k' + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If only a cut with the scope $m$ was evaluated, the corresponding derivation has the length $\ell = i + \ell' + 2$. Thus, we obtain

$$k + 1 = i + k' + 1 \leq i + \ell' < i + \ell' + 2 = \ell$$

and

$$\ell = i + \ell' + 2 \leq i + 3 \cdot k' + 3 \overset{i>0}{\leq} 3 \cdot i + 3 \cdot k' + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

- If the ISO execution raises an error which is not caught or evaluates a halting predicate, so does the derivation. Then we again have $i = j$. From the derivation obtained by the induction hypothesis we construct the corresponding derivation for the initial goal continuing from the state $\langle (B'\sigma, Q'\sigma)_{\sigma,\varepsilon} \mid (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ by first replacing all answer substitutions and candidate answer substitutions $\delta$ in the obtained derivation by $\sigma\delta$. Then we replace all those $!_0$ by $!_m$ which have been at some position in $(B\sigma, Q\sigma)$ where there is a $!_m$ at the same position in $(B'\sigma, Q'\sigma)$ (again assuming that the scope $m$ is not used within the derivation). Afterwards we add the goals $(t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m$ before the initial scope marker in every state up to the first state where a cut with the initial scope or the scope $m$ is executed. If the evaluated cut has the scope $m$, then we add the goal $?_m$ before the initial scope marker to all other states up to the first one where a cut with the initial scope is evaluated. Note that these changes do not modify the length of the obtained derivation. Also note that the execution is finished after $i + k'$ unification tests as the execution is terminated by the error or the halting predicate (due to our assumptions). Since the constructed derivation is also finished, it has the length $\ell = i + \ell' + 1$. As we have $i > 0$, we obtain

$$k = i + k' \leq i + \ell' < i + \ell' + 1 = \ell$$

and

$$\ell = i + \ell' + 1 \leq i + 3 \cdot k' + 2 \overset{i>0}{<} 3 \cdot i + 3 \cdot k' + 1 = 3 \cdot k + 1.$$

The derivation leads to an error iff the ISO execution does. Thus, the proposition holds in this case.

- If the ISO execution neither reaches a cut as described above nor raises an uncaught error or evaluates a halting predicate, so does the derivation. From the derivation obtained by the induction hypothesis we construct a derivation from the state $\langle (B'\sigma, Q'\sigma)_{\sigma,\varepsilon} \mid (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon\rangle$ to a state $\langle (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A'\rangle$ where $\mathcal{D}'$ is the dynamic part of the program, $\mathcal{E}'$ is the environment, $\mathcal{PI}'$ is the set of user-defined predicate indicators in the last state of the obtained derivation, and $A'$ is constructed from the list of answer substitutions $A$ in that last state by replacing every $\delta \in A$ by $\sigma\delta$. This new derivation is constructed by first replacing all answer substitutions and candidate answer substitutions $\delta$ in the obtained derivation by $\sigma\delta$. Then we replace all those $!_0$ by $!_m$ which have been at some position in $(B\sigma, Q\sigma)$ where there is a $!_m$ at the same position in $(B'\sigma, Q'\sigma)$ (again assuming that the scope $m$ is not used within the derivation). Afterwards we add the goals $(t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m$ before the initial scope marker to every state. Finally, we drop the last application of the (Failure) rule from the derivation. Thus, the length of the derivation to the state $\langle (t', Q')^{c'_{i+1}}_{\varnothing,\varepsilon} \mid \ldots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A'\rangle$ is reduced by one compared to the derivation obtained by the induction hypothesis. The execution took $i + k'$ steps up to this point, while the constructed derivation so far has the length $i + \ell'$. Now we have reached a situation which is similar to the one before executing the first unification test. But we have one child node less to consider in the execution and at least one goal less in the derivation. Hence, we can again use the same reasoning. The only additional change in the continuation of the derivation is that the already existing answer substitutions in $A'$ are added to the new answer substitution list without any changes in the beginning of that list. This reasoning cannot be repeated more often than $n$ times. The execution takes $k = n' + \Sigma_{a=1}^{n''} k_a$ unification tests where $n'$ is the number of necessary unification tests for the execution of $t$, $n''$ is the number of successful necessary unification tests for the execution of $t$ and $k_a$ is the number of unification tests performed for the initial node's child nodes for all $a \in \{1, \ldots, n''\}$ (with $k' = k_1$). If the ISO execution results in an error, so does the corresponding derivation. Then the length of the corresponding derivation is $\ell = 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)$ where $\ell_a$ is the length of the corresponding derivation for the $k_a$ unification tests for each $a \in \{1, \ldots, n''\}$. The reason for this length is that for each sub-derivation the last application of the (Failure) rule is dropped except for the last sub-derivation. Together with the initial application of the (Case) rule and the $n'$ applications of the (Eval) and (Backtrack) rules, this yields the described length. Moreover, we have $k_a + 1 \leq \ell_a \leq 3 \cdot k_a + 1$ for each $a \in \{1, \ldots, n'' - 1\}$ and $k_{n''} \leq \ell_{n''} \leq 3 \cdot k_{n''} + 1$. Now we obtain

$$\begin{aligned} k &= n' + \Sigma_{a=1}^{n''} k_a \\ &\leq 1 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &< 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &\leq 2 + n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ &\overset{n'>0}{<} 1 + 3 \cdot n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ &= 3 \cdot k + 1. \end{aligned}$$

Hence, the proposition holds in this case. If a halting predicate is evaluated, we use the same construction and obtain the same length and, thus, the proposition holds again in this case. Otherwise there are two cases depending on whether a cut with the initial scope has been evaluated. If this is the case we finally reach the state $\langle ?_0 \ ; \ \mathcal{D}'' \ ; \ \mathcal{E}'' \ ; \ \mathcal{PI}'' \ ; \ A'' \rangle$ for the dynamic part $\mathcal{D}''$ of the program, the environment $\mathcal{E}''$, the user-defined predicate indicator set $\mathcal{PI}''$, and the list of answer substitutions $A''$ obtained by the complete execution. The derivation can be finished by one final application of the (Failure) rule. Then we have $\ell = 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)$ again. We obtain

$$\begin{aligned} k + 1 &= 1 + n' + \Sigma_{a=1}^{n''} k_a \\ &\leq 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{aligned}\ell &= 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)\\ &\leq 2 + n' + \Sigma_{a=1}^{n''} 3 \cdot k_a\\ &\overset{n'>0}{<} 1 + 3 \cdot n' + \Sigma_{a=1}^{n''} 3 \cdot k_a\\ &= 3 \cdot k + 1.\end{aligned}$$

Hence, the proposition holds in this case. If no cut with the initial scope was evaluated, we finally reach the state $\langle ?_m \mid ?_0 ~;~ \mathcal{D}'' ~;~ \mathcal{E}'' ~;~ \mathcal{PI}'' ~;~ A''\rangle$. The derivation can be finished by two final applications of the (Failure) rule. Then we have $\ell = 3 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)$. Thus, we obtain

$$\begin{aligned}k + 1 &= 1 + n' + \Sigma_{a=1}^{n''} k_a\\ &\leq 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)\\ &< 3 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)\\ &= \ell\end{aligned}$$

and

$$\begin{aligned}\ell &= 3 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)\\ &\leq 3 + n' + \Sigma_{a=1}^{n''} 3 \cdot k_a\\ &\overset{n'>0}{\leq} 1 + 3 \cdot n' + \Sigma_{a=1}^{n''} 3 \cdot k_a\\ &= 3 \cdot k + 1.\end{aligned}$$

Hence, the proposition also holds in this case.

– If $root(t)$ is a built-in predicate raising an error, then the ISO execution continues by evaluating the built-in predicate throw/1 leading to a program error. The corresponding derivation consists of one application of the (Error) rule followed by one application of the (ThrowErr) rule. Since we have $k = \ell = 2$ for the length $\ell$ of the derivation, the proposition holds in this case. For all further cases, we can, thus, assume that the respective built-in predicate does not raise an error itself.
– If $root(t)$ is the built-in predicate =/2, then we have $t = {}'{=}'(t_1, t_2)$ and the first unification test is performed between $t_1$ and $t_2$. Moreover, we have $t_1 \sim t_2$, because the execution takes more than one unification test. Let $mgu(t_1, t_2) = \sigma$. The execution creates a new node labeled with $Q\sigma$ and $\sigma$ which is the new current node. The derivation starts with one application of the (UnifySuccess) rule reaching the state $\langle (Q\sigma)_{\sigma,\varepsilon} \mid ?_0 ~;~ \mathcal{D} ~;~ \mathcal{E} ~;~ \mathcal{PI} ~;~ \varepsilon\rangle$. For the execution of the goal $Q\sigma$, we can use the induction hypothesis to obtain a corresponding derivation, because the execution of that goal takes $k' < k$ unification tests. If the execution results in an error, so does the derivation and we obtain $k' \leq \ell' \leq 3 \cdot k' + 1$ for the length $\ell'$ of the obtained derivation. We construct a derivation for the initial goal by replacing every

answer substitution or candidate answer substitution $\delta$ with $\sigma\delta$ in the obtained derivation, and appending this derivation to the first application of the (UnifySuccess) rule. The ISO execution takes $k = k' + 1$ unification tests while the length of the derivation is $\ell = \ell' + 1$. Thus, we have

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. Otherwise we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain the corresponding derivation for the initial goal in the same way. Dropping the last application of the (Failure) rule is not necessary since it is also applied in the derivation for the initial goal. Thus, we have $\ell = \ell' + 1$ and obtain

$$k + 1 = k' + 2 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate atom_chars/2, atom_codes/2, atom_length/2, char_code/2, number_chars/2, number_codes/2, get_byte/2, get_byte/1, peek_byte/2, peek_byte/1, number_codes/2, get_char/2, get_char/1, get_code/2, get_code/1, peek_char/2, peek_char/1, peek_code/2, peek_code/1, at_end_of_stream/1, at_end_of_stream/0, current_input/1, current_output/1, arg/3, copy_term/2, functor/3, =../2, \=/2, unify_with_occurs_check/2, ==/2, \==/2, @>/2, @>=/2, @</2, @=</2, atom/1, atomic/1, compound/1, float/1, integer/1, nonvar/1, number/1, var/1, read_term/3, read_term/2, read/2, or read/1, then the proof is analogous to the case for =/2.
– If $root(t)$ is the built-in predicate is/2, then we have $t = \mathsf{is}(t_1, t_2)$ and the unification test is performed between $t_1$ and $evaluate(t_2)$ where $evaluate(t_2) \in \mathbb{F} \cup \mathbb{I}$. Moreover, we have $t_1 \sim evaluate(t_2)$ since otherwise the execution would only take one unification test. Let $mgu(t_1, evaluate(t_2)) = \sigma$. The execution creates a new node labeled with $Q\sigma$ and $\sigma$ which is the new current node. The derivation starts with one application of the (IsSuccess) rule reaching the state $\langle (Q\sigma)_{\sigma,\varepsilon} \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$. For the execution of the goal $Q\sigma$, we can use the induction hypothesis to obtain a corresponding derivation, because the execution of the goal takes $k' < k$ unification tests. If the execution of $Q\sigma$ results in an error, so does the derivation and we obtain $k' \leq \ell' \leq 3 \cdot k' + 1$ for the length $\ell'$ of the obtained derivation. We construct a derivation for the initial goal by replacing every answer substitution or candidate answer substitution $\delta$ with $\sigma\delta$ in the obtained derivation, and appending this derivation to the first application of the (IsSuccess) rule. The ISO execution takes $k = k' + 1$ unification tests while the length of the derivation is $\ell = \ell' + 1$. Thus, we have

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the ISO execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, then we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain the corresponding derivation for the initial goal in the same way. Dropping the last application of the (Failure) rule is not necessary since it is also applied in the derivation for the initial goal. Thus, we have $\ell = \ell' + 1$ and obtain

$$k + 1 = k' + 2 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate $\mathsf{asserta}/1$, then we have $t = \mathsf{asserta}(c)$. If $root(c) = :-/2$, then we have $c = h' :- B'$ and define $h = h'$ and $B = B'[\mathcal{V}/\mathsf{call}(\mathcal{V})]$. Otherwise let $h = c$ and $B = \mathsf{true}$. Then we have that $h$ is no variable, $root(h)$ is not a static predicate and $B$ is a well-formed query. The clause $h :- B$ is inserted before all other clauses for the same predicate in the program and the execution continues with the query $Q$ and the empty local substitution. Since this $\mathsf{ISO}$ execution takes $k' < k$ unification tests, we can use the induction hypothesis to obtain a corresponding derivation for this execution of the length $\ell'$. We construct the corresponding derivation for the initial goal by appending the obtained derivation to one application of the (AssA) rule where $h :- B$ is added to $\mathcal{D}$ before all other clauses (and, thus, definitely before all other clauses for the same predicate) and the predicate indicator $PI$ of $h$ is added to the set $\mathcal{PI}$. Then the execution takes $k = k' + 1$ steps and the derivation has the length $\ell = \ell' + 1$. If the execution results in an error, so does the derivation and we have $k' \leq \ell' \leq 3 \cdot k' + 1$. Thus, we obtain

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. Otherwise we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain

$$k + 1 = k' + 2 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Again, the proposition holds in this case.

– If $root(t)$ is the built-in predicate $\mathsf{assertz}/1$, then the proof is analogous to the case for $\mathsf{asserta}/1$.

– If $root(t)$ is the built-in predicate $\mathsf{retract}/1$, then we have $t = \mathsf{retract}(c)$. If $root(c) = :-/2$, then we have $c = h :- B$. Otherwise let $h = c$ and $B = \mathsf{true}$. Moreover, we have that $h$ is no variable, $root(h)$ is not a static predicate, and there is at least one clause in the program for $root(h)$. The unification test is performed between $h :- B$ and the first clause for $root(h)$ in the program. If there is only one clause $h' :- B'$ for $root(h)$ in the program, then we must have $h :- B \sim h' :- B'$, because we have $k > 1$. Let $\sigma = mgu(h :- B, h' :- B')$. Then the $\mathsf{ISO}$ execution creates a new node labeled with $Q\sigma$ and $\sigma$ and removes the clause $h' :- B'$ from the program. As the execution of $Q\sigma$ takes $k' = k - 1$ unification tests, we can use the induction hypothesis to obtain a corresponding derivation for this execution. If it results in an error, so does the derivation and for its length $\ell'$ we have $k' \leq \ell' \leq 3 \cdot k' + 1$. We obtain the derivation for the initial goal by replacing every answer substitution or candidate answer substitution $\delta$ in the obtained derivation with $\sigma\delta$ and by appending this derivation to the first two applications of the (Retract) and (RetSuc) rules. Then the derivation for the initial goal has the length $\ell = \ell' + 2$. Thus, we obtain

$$k = k' + 1 \leq \ell' + 1 < \ell' + 2 = \ell \leq 3 \cdot k' + 3 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Since the program after applying the (RetSuc) rule is the same as for the execution after evaluating the retract predicate, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, then we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain the corresponding derivation in the same way as before. We obtain

$$k + 1 = k' + 2 \leq \ell' + 1 < \ell' + 2 = \ell \leq 3 \cdot k' + 3 < 3 \cdot k' + 4 = 3 \cdot k + 1$$

and, thus, the proposition holds in this case, too. If there is more than one clause for $root(h)$ in the program, then let $Slice_{\mathcal{D}}(t) = (h_1 :- B_1, \ldots, h_n :- B_n)$. The corresponding derivation starts with one application of the (Retract) rule. Let $i \in \{1, \ldots, n\}$ be the first index such that $h :- B \sim h_i :- B_i$. If no such index exists, then the ISO execution consists of $n$ failing unification tests and is finished afterwards. The corresponding derivation continues with $n$ applications of the (RetFail) rule followed by one application of the (Failure) rule reaching the empty list of goals with the empty list of answer substitutions. The execution takes $k = n$ unification tests while the derivation has the length $\ell = n + 2$. We obtain

$$k + 1 = n + 1 < n + 2 = \ell \overset{n>1}{<} 3 \cdot n + 1 = 3 \cdot k + 1$$

and, thus, the proposition holds in this case. If an index $i$ as described above exists, then the execution starts with $i - 1$ failing unification tests followed by one succeeding unification test leading to a new node labeled with $Q\sigma_i$ and $\sigma_i$ where $mgu(h :- B, h_i :- B_i) = \sigma_i$. Furthermore, the clause $h_i :- B_i$ is removed from the program yielding the dynamic part $\mathcal{D}'$ of the program. The corresponding derivation continues with $i - 1$ applications of the (RetFail) rule followed by one application of the (RetSuc) rule reaching the state $\langle (Q\sigma_i)_{\sigma_i,\varepsilon} \mid :\not-^{h :- B,Q,\varnothing}_{(c_{i+1},m_{i+1})} \mid \cdots \mid :\not-^{h :- B,Q,\varnothing}_{(c_n,m_n)} \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$. For the execution of $Q\sigma_i$ we can use the induction hypothesis to obtain a corresponding derivation as this execution takes $k_1 < k$ unification tests. If this execution leads to a program error, so does the derivation. Then we have $k_1 \leq \ell_1 \leq 3 \cdot k_1 + 1$ for the length $\ell_1$ of the obtained derivation. Then the ISO execution is finished due to the error. We construct the corresponding derivation for the initial goal by replacing every answer substitution and candidate answer substitution $\delta$ in the obtained derivation by $\sigma_i\delta$. Then we add the goals $:\not-^{h :- B,Q,\varnothing}_{(c_{i+1},m_{i+1})} \mid \cdots \mid :\not-^{h :- B,Q,\varnothing}_{(c_n,m_n)}$ before the initial scope marker to every state in the obtained derivation up to the first state where a cut with the initial scope is evaluated. Finally, we append this derivation to the first $i+1$ rule applications leading to the state $\langle (Q\sigma_i)_{\sigma_i,\varepsilon} \mid {:\!\not-}^{\,h\,:\text{-}\,B,Q,\varnothing}_{(c_{i+1},m_{i+1})} \mid \cdots \mid {:\!\not-}^{\,h\,:\text{-}\,B,Q,\varnothing}_{(c_n,m_n)} \mid ?_0 \;;\; \mathcal{D}' \;;\; \mathcal{E} \;;\; \mathcal{PI} \;;\; \varepsilon\rangle$. Both execution and derivation result in an error and reach the same program. Moreover, the derivation for the initial goal has the length $\ell = \ell_1 + i + 1$ while the execution takes $k = k_1 + i$ unification tests and, thus, we obtain

$$k = k_1 + i \leq \ell_1 + i < \ell_1 + i + 1 = \ell$$

and

$$\ell = \ell_1 + i + 1 \leq 3 \cdot k_1 + i + 2 \overset{i>0}{<} 3 \cdot k_1 + 3 \cdot i + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated during the execution of $Q\sigma_i$, we have $k_1 + 1 \leq \ell_1 \leq 3 \cdot k_1 + 1$. If a cut with the initial scope is evaluated during the execution of $Q\sigma_i$, then the execution of the initial goal is also finished as all unvisited nodes have been deleted due to the cut. We construct the derivation for the initial goal in the same way as before and, thus, obtain

$$k + 1 = k_1 + i + 1 \leq \ell_1 + i < \ell_1 + i + 1 = \ell$$

and

$$\ell = \ell_1 + i + 1 \leq 3 \cdot k_1 + i + 2 \overset{i>0}{<} 3 \cdot k_1 + 3 \cdot i + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If no error occurs, no halting predicate is evaluated, and no cut with the initial scope is evaluated during the execution of $Q\sigma_i$, then we continue the derivation for the initial goal in the same way as before except that we drop the last application of the (FAILURE) rule from the derivation obtained for the execution of $Q\sigma_i$. Hence, the length of the derivation up to this point is $\ell_1 + i$ while the ISO execution takes $k_1 + i$ unification tests so far. Now we have reached a situation which is similar to the one before executing the first unification test. But we have one child node less to consider in the execution and at least one goal less in the derivation. Hence, we can again use the same reasoning. There are three additional changes in the continuation of the derivation. First, the already existing answer substitutions in $A'$ are added to the new answer substitution list without any changes in the beginning of that list. Second, the remaining execution may also have only one unification test. Third, the clauses which have to be removed from the program eventually may already be removed from the program (this is also true if another identical clause has been inserted afterwards, such that there exists an identical clause in the program). If the respective clause has already been removed before, neither the re-execution of retract, nor the evaluation of the (RETSUC) rule change the current program. This reasoning cannot be repeated more often than $n$ times. The ISO execution takes $k = n' + \Sigma_{a=1}^{n''} k_a$ unification tests where $n'$ is the number of necessary unification tests for the execution of $\mathsf{retract}(c)$, $n''$ is the number of successful necessary unification tests for this execution, and $k_a$ is the number of unification tests performed for the initial node's child nodes for all $a \in \{1, \ldots, n''\}$. If the execution results in an error, so does the corresponding derivation. Then the length of the corresponding derivation is $\ell = 2+n'+\Sigma_{a=1}^{n''}(\ell_a-1)$ where $\ell_a$ is the length of the corresponding derivation for the $k_a$ unification tests for each $a \in \{1, \ldots, n''\}$. The reason for this length is that for each sub-derivation, the last application of the (Failure) rule is dropped except for the last sub-derivation. Together with the initial application of the (Retract) rule and the $n'$ applications of the (RetSuc) and (RetFail) rules, this yields the described length. Moreover, we have $k_a+1 \leq \ell_a \leq 3 \cdot k_a+1$ for each $a \in \{1, \ldots, n''-1\}$ and $k_{n''} \leq \ell_{n''} \leq 3 \cdot k_{n''}+1$. Now we obtain

$$\begin{aligned} k &= n' + \Sigma_{a=1}^{n''} k_a \\ &\leq 1 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &< 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &\leq 2 + n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ &\overset{n'>0}{<} 1 + 3 \cdot n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ &= 3 \cdot k + 1. \end{aligned}$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. Otherwise we finally reach the state $\langle ?_0 \; ; \; \mathcal{D}'' \; ; \; \mathcal{E}'' \; ; \; \mathcal{PI}'' \; ; \; A'' \rangle$ for the dynamic part $\mathcal{D}''$ of the program, the environment $\mathcal{E}''$, the set of user-defined predicate indicators $\mathcal{PI}''$, and the list of answer substitutions $A''$ obtained by the complete execution. The derivation can be finished by one final application of the (Failure) rule. Then we have $\ell = 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1)$ again. We obtain

$$\begin{aligned} k + 1 &= 1 + n' + \Sigma_{a=1}^{n''} k_a \\ &\leq 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{array}{rcl} \ell & = & 2 + n' + \Sigma_{a=1}^{n''}(\ell_a - 1) \\ & \leq & 2 + n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ & \overset{n'>0}{<} & 1 + 3 \cdot n' + \Sigma_{a=1}^{n''} 3 \cdot k_a \\ & = & 3 \cdot k + 1. \end{array}$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate $\mathsf{findall}/3$, then we have $t = \mathsf{findall}(r, g, s)$ where $g$ is callable and $s$ is a variable, partial list, or a list. The execution continues with the sub-computation of the goal $\mathsf{call}(g)$. By the induction hypothesis we obtain a corresponding derivation for this sub-computation as it will take $k' < k$ unification tests. Let $\ell'$ be the length of the obtained derivation. If the sub-computation results in an error, so does the derivation. Then we have $k' \leq \ell' \leq 3 \cdot k' + 1$. We construct a derivation for the initial goal by adding the goal $\%^{r,[],s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker $?_0$ to all states in that derivation up to the first application of the (Success) rule. This application is replaced by an application of the (FindNext) rule and in all following states we add the goal $\%^{r,[r\delta],s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker where $\delta$ is the answer substitution obtained for the first success. We repeat this construction for all applications of the (Success) rule. Finally, we append this derivation to the first application of the (Findall) rule. Since the execution takes $k = k' + 1$ unification tests and the derivation has the length $\ell = \ell' + 1$, we obtain

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If the sub-computation neither results in an error nor evaluates a halting predicate, then we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$. After the sub-computation is done with answer substitutions $\delta_1, \ldots, \delta_m$ for some $m \geq 0$, the execution continues with a unification test of $s$ and the list $[r\delta_1, \ldots, r\delta_m]$. We construct a derivation for the initial goal by starting with one application of the (Findall) rule reaching the state $\langle \mathsf{call}(g)_{\varnothing,\varepsilon} \mid \%^{r,[],s}_{Q,\varnothing,\varepsilon} \mid ?_0 \; ; \; \mathcal{D} \; ; \; \mathcal{E} \; ; \; \mathcal{PI} \; ; \; \varepsilon\rangle$. Then we take the derivation obtained for the sub-computation of $\mathsf{call}(g)$ and add the goal element $\%^{r,[],s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker to all states in that derivation up to the first application of the (Success) rule. This application is replaced by an application of the (FindNext) rule and in all following states we add the goal $\%^{r,[r\delta],s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker where $\delta$ is the answer substitution obtained for the first success. We repeat this construction for all applications of the (Success) rule. Finally, we drop the last application of the (Failure) rule. As the derivation for the sub-computation has the same answer substitutions as the ISO execution, the constructed derivation ends in the state $\langle \%^{r,[r\delta_1,\ldots,r\delta_m],s}_{Q,\varnothing,\varepsilon} \mid ?_0 \; ; \; \mathcal{D}' \; ; \; \mathcal{E}' \; ; \; \mathcal{PI}' \; ; \; \varepsilon\rangle$ where $\mathcal{D}'$ is the dynamic part of the program, $\mathcal{E}'$ is the environment, and $\mathcal{PI}'$ is the set of user-defined predicate indicators after the execution of the sub-computation. Then we apply the (FoundAll) rule and reach the state $\langle ([r\delta_1, \ldots, r\delta_m] = s, Q)_{\varnothing,\varepsilon} \mid ?_0 \; ; \; \mathcal{D}' \; ; \; \mathcal{E}' \; ; \; \mathcal{PI}' \; ; \; \varepsilon\rangle$. If $mgu(s, [r\delta_1, \ldots, r\delta_m]) = \mathit{fail}$, then the execution is finished since the set of unvisited nodes is empty. The corresponding derivation continues with one application of the (UnifyFail) rule followed by one application of the (Failure) rule reaching the state $\langle \varepsilon \; ; \; \mathcal{D}' \; ; \; \mathcal{E}' \; ; \; \mathcal{PI}' \; ; \; \varepsilon\rangle$ where the derivation is also finished and has, thus, the length $\ell = \ell' + 3$ while the number of unification tests is $k = k' + 2$. Now we obtain

$$k = k' + 2 \leq \ell' + 1 < \ell' + 3 = \ell \leq 3 \cdot k' + 4 < 3 \cdot k' + 7 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If $s \sim [r\delta_1, \ldots, r\delta_m]$, let $mgu(s, [r\delta_1, \ldots, r\delta_m]) = \sigma$. The execution continues by creating a new node labeled with $Q\sigma$ and $\sigma$ which is the new current node. The derivation continues with one application of the (UnifySuccess) rule reaching the state $\langle (Q\sigma)_{\sigma,\varepsilon} \mid ?_0 \; ; \; \mathcal{D}' \; ; \; \mathcal{E}' \; ; \; \mathcal{PI}' \; ; \; \varepsilon\rangle$. For the execution of the goal $Q\sigma$, we can use the induction hypothesis to obtain a corresponding derivation, because the ISO execution of the goal takes $k'' < k$ unification tests. Let $\ell''$ be the length of the corresponding derivation for the execution of $Q\sigma$. The whole execution now takes $k = k' + k'' + 2$ unification tests. If the execution of $Q\sigma$ results in an error, we have $k'' \leq \ell'' \leq 3 \cdot k'' + 1$ and the derivation results in an error, too. We continue the construction of the derivation for the initial goal by replacing every answer substitution or candidate answer substitution $\delta$ in the obtained derivation for $Q\sigma$ with $\sigma\delta$ and appending this derivation to the application of the (UnifySuccess) rule for $[r\delta_1, \ldots, r\delta_m] = s$. Then the length of the derivation is $\ell = \ell' + \ell'' + 2$. Thus, we have

$$k = k' + k'' + 2 \leq \ell' + \ell'' + 1 < \ell' + \ell'' + 2 = \ell$$

and

$$\ell = \ell' + \ell'' + 2 \leq 3 \cdot k' + 3 \cdot k'' + 4 < 3 \cdot k' + 3 \cdot k'' + 7 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated during the execution, we obtain the derivation for the initial goal in the same way as before since we do not have to drop the final application of the (Failure) rule. Thus, we have $\ell = \ell' + \ell'' + 2$ again, but this time we also have $k'' + 1 \leq \ell'' \leq 3 \cdot k'' + 1$. We obtain

$$k + 1 = k' + k'' + 3 \leq \ell' + \ell'' + 1 < \ell' + \ell'' + 2 = \ell$$

and

$$\ell = \ell' + \ell'' + 2 \leq 3 \cdot k' + 3 \cdot k'' + 4 < 3 \cdot k' + 3 \cdot k'' + 7 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate bagof/3, then we have $t = \mathsf{bagof}(r, b, s)$ where $b = {}^\wedge(t_1, {}^\wedge(t_2, {}^\wedge(\ldots, {}^\wedge(t_m, g)\ldots)))$ for some $m \geq 0$, $root(g) \neq {}^\wedge/2$, $g$ is a callable term, and $s$ is a variable, a list, or a partial list. The execution continues with the computation of the goal $\mathsf{findall}([f(X_1, \ldots, X_n), f], g, Y)$ where $\mathcal{V}(g) \setminus (\bigcup_{i=1}^{m} \mathcal{V}(t_i)) = \{X_1, \ldots, X_n\}$, $f/n \in \Sigma$ is fresh, and $Y \in \mathcal{V}$ is fresh. The execution further continues with the sub-computation of the goal $\mathsf{call}(g)$. By the induction hypothesis we obtain a corresponding derivation for this sub-computation as it will take $k' < k$ unification tests. The obtained derivation has the length $\ell'$. If the sub-computation results in an error, so does the derivation and we have $k' \leq \ell' \leq 3 \cdot k' + 1$. Then we construct a derivation for the initial goal by adding the goals $\%^{[f(X_1,\ldots,X_n),r],[\,],Y}_{\square,\varnothing,\varepsilon} \mid \$^{s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker to all states in that derivation up to the first application of the (Success) rule. This application is replaced by an application of the (FindNext) rule and in all following states we add the goals $\%^{[f(X_1,\ldots,X_n),r],[[f(X_1,\ldots,X_n),r]\delta],Y}_{\square,\varnothing,\varepsilon} \mid \$^{s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker where $\delta$ is the answer substitution obtained for the first success. We repeat this construction for all applications of the (Success) rule while we leave the last (error) state unchanged. We append this derivation to the first two applications of the (Bagof) and (Findall) rules. Since the execution takes $k = k' + 2$ steps and the derivation has the length $\ell = \ell' + 2$, we obtain

$$k = k' + 2 \leq \ell' + 2 = \ell \leq 3 \cdot k' + 3 < 3 \cdot k' + 7 = 3 \cdot k + 1.$$

So the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. Otherwise we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$. After the sub-computation is done, a unification test between $Y$ and a list $[[w_1, r_1], \ldots, [w_m, r_m]]$ for some $m \geq 0$ is performed. Since $Y$ is fresh, this test succeeds. We construct the corresponding derivation for the initial goal by adding the goals $\%^{[f(X_1,\ldots,X_n),r],[\,],Y}_{\square,\varnothing,\varepsilon} \mid \$^{s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker to all states in that derivation up to the first application of the (Success) rule. This application is replaced by an application of the (FindNext) rule and in all following states we add the goals $\%^{[f(X_1,\ldots,X_n),r],[[f(X_1,\ldots,X_n),r]\delta],Y}_{\square,\varnothing,\varepsilon} \mid \$^{s}_{Q,\varnothing,\varepsilon}$ before the initial scope marker where $\delta$ is the answer substitution obtained for the first success. We repeat this construction for all applications of the (Success) rule. We append this derivation to the first two applications of the (Bagof) and (Findall) rules and drop the last application of the (Failure) rule. Then the derivation continues with one application of the (FoundBag) rule reaching the state $\langle \$^{s,[[w_1,r_1],\ldots,[w_m,r_m]]}_{Q,\varnothing,\varepsilon} \mid ?_0 \;;\; \mathcal{D}' \;;\; \mathcal{E}' \;;\; \mathcal{PI}' \;;\; \varepsilon\rangle$ for the dynamic part $\mathcal{D}'$ of the program, the environment $\mathcal{E}'$, and the set $\mathcal{PI}'$ of user-defined predicate indicators reached after the sub-computation. If $[[w_1, r_1], \ldots, [w_m, r_m]]$ is empty, the computation is finished since the set of unvisited nodes is empty. The corresponding derivation continues with one application of the (EmptyBag) rule followed by one application of the (Failure) rule reaching the state $\langle \varepsilon \;;\; \mathcal{D}' \;;\; \mathcal{E}' \;;\; \mathcal{PI}' \;;\; \varepsilon\rangle$ where the derivation is finished, too. The execution takes $k = k' + 3$ unification tests and the derivation has the length $\ell = \ell' + 4$.

We obtain

$$k+1=k'+4\le \ell'+3<\ell'+4=\ell\le 3\cdot k'+5<3\cdot k'+10=3\cdot k+1.$$

Hence, the proposition holds in this case. Otherwise the execution continues by choosing one element $[w,q]$ in $[[w_1,r_1],\dots,[w_m,r_m]]$ (which one is implementation-defined) which has not been used yet. Then let $s'$ be the list of all elements $[w',q']$ in $[[w_1,r_1],\dots,[w_m,r_m]]$ where $w'$ is a variant of $w$ (the order of the elements in $s'$ is the same as in n $[[w_1,r_1],\dots,[w_m,r_m]]$. Then the next unification test is a unification test between $s\sigma$ and $s''$ where $s''$ is the list of all $q'\sigma$ such that there is some $w'$ where $[w',q']$ is an element of $s'$ and $\sigma$ is the *mgu* of all $w'$ such that there is some $q'$ where $[w',q']$ is an element of $s'$. The order of the elements in $s''$ is the same as in $s'$. If $mgu(s\sigma,s'')=\mathit{fail}$, then another element from the list $s'''$ is chosen and we repeat the construction of a sublist of $s'$ and a corresponding unification test where $s'''$ is the list $[[w_1,r_1],\dots,[r_m,r_m]]$ from which we removed all elements of $s'$. The corresponding derivation continues with one application of the (NextBag) rule followed by one application of the (UnifyFail) rule. Otherwise, if $s\sigma\sim s''$, then the execution continues with a new node labeled with $Q\sigma\sigma'$ and local substitution $\sigma\sigma'$ where $mgu(s\sigma,s'')=\sigma'$. For the further execution of this node we can again use the induction hypothesis to obtain a corresponding derivation since the execution takes $k_1<k$ unification tests. Let $\ell_1$ be the length of the obtained derivation. We continue the derivation for the initial goal by one application of the (NextBag) rule reaching the state $\langle (s\sigma=s'',Q\sigma)_{\sigma,\varepsilon}\mid \$^{s,s'''}_{Q,\varnothing,\varepsilon}\mid ?_0\ ;\ \mathcal{D}'\ ;\ \mathcal{E}'\ ;\ \mathcal{PI}'\ ;\ \varepsilon\rangle$. Then we apply the (UnifySuccess) rule and reach the state $\langle (Q\sigma\sigma')_{\sigma\sigma',\varepsilon}\mid \$^{s,s'''}_{Q,\varnothing,\varepsilon}\mid ?_0\ ;\ \mathcal{D}'\ ;\ \mathcal{E}'\ ;\ \mathcal{PI}'\ ;\ \varepsilon\rangle$. Finally, we append the derivation obtained for the execution of $Q\sigma\sigma'$ where we added the goal $\$^{s,s'''}_{Q,\varnothing,\varepsilon}$ before the initial scope marker to all states in that derivation up to the first one where a cut with the initial scope is evaluated. If no such state exists, we add the goal to all states except the last one. If the execution of $Q\sigma\sigma'$ results in an error, evaluates a halting predicate, or evaluates a cut with the initial scope, we are done. Then the execution takes $k=k'+k_1+4$ unification tests while the derivation has the length $\ell=\ell'+\ell_1+4$. If the execution results in an error, we have $k_1\le\ell_1\le 3\cdot k_1+1$. Thus, we obtain

$$k=k'+k_1+4\le \ell'+\ell_1+3<\ell'+\ell_1+4=\ell$$

and

$$\ell=\ell'+\ell_1+4\le 3\cdot k'+3\cdot k_1+6<3\cdot k'+3\cdot k_1+13=3\cdot k+1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurred and no halting predicate is evaluated so far, but we evaluated a cut with the initial scope, we have $k_1+1\le\ell_1\le 3\cdot k_1+1$. Thus, we obtain

$$k+1=k'+k_1+5\le \ell'+\ell_1+3<\ell'+\ell_1+4=\ell$$

and

$$\ell = \ell' + \ell_1 + 4 \leq 3 \cdot k' + 3 \cdot k_1 + 6 < 3 \cdot k' + 3 \cdot k_1 + 13 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case, too. If neither an error occurs nor a halting predicate or a cut with the initial scope is evaluated, then we drop the last application of the (Failure) rule. Up to this point the ISO execution takes $k' + k_1 + 4$ unification tests while the derivation has a length of $\ell' + \ell_1 + 3$ and $s'''$ contains exactly the unused elements of $s'$. Moreover, we have $k_1 + 1 \leq \ell_1 \leq 3 \cdot k_1 + 1$. We continue this construction for all re-executions of $\mathsf{bagof}(r, b, s)$ as long as no error occurs and no halting predicate or a cut with initial scope is evaluated. Then the execution takes $k = k' + n + 3 + \Sigma_{i=1}^{n'} k_i$ unification tests where $n$ is the number of unification tests for the (re-)executions of $\mathsf{bagof}(r, b, s)$ after executing the query $\mathsf{findall}([f(X_1, \ldots, X_n), r], g, Y)$, $n'$ is the number of successful such unification tests, and $k_i$ is the number of unification tests performed for the respective child nodes of $\mathsf{bagof}(r, b, s)$ for all $i \in \{1, \ldots, n'\}$. If the execution results in an error or evaluates a halting predicate or a cut with the initial scope, the corresponding derivation has the length $\ell = \ell' + 2 \cdot n - n' + 3 + \Sigma_{i=1}^{n'} \ell_i$ where $\ell_i$ is the length of the derivation obtained for the execution of the $i$-th child of the node $\mathsf{bagof}(r, b, s)$ for all $i \in \{1, \ldots, n'\}$ (the derivation for the first findall execution plus two rule applications to reach it minus one dropped (Failure) rule plus one to reach the further execution of bagof, two rule applications for each unification test in the (re-)execution of bagof, dropping the last (Failure) rule for the executions after all successful unifications except the last one and the derivations for the children of the bagof node). Moreover, we have $k_i + 1 \leq \ell_i \leq 3 \cdot k_i + 1$ for all $i \in \{1, \ldots, n' - 1\}$ and $k_{n'} \leq \ell_{n'} \leq 3 \cdot k_{n'} + 1$. Thus, we obtain

$$\begin{aligned} k &= k' + n + 3 + \Sigma_{i=1}^{n'} k_i \\ &\leq \ell' + n - n' + 3 + \Sigma_{i=1}^{n'} \ell_i \\ &< \ell' + 2 \cdot n - n' + 3 + \Sigma_{i=1}^{n'} \ell_i \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= \ell' + 2 \cdot n - n' + 3 + \Sigma_{i=1}^{n'} \ell_i \\ &\leq 3 \cdot k' + 2 \cdot n + 4 + 3 \cdot \Sigma_{i=1}^{n'} k_i \\ &< 3 \cdot k' + 3 \cdot n + 10 + 3 \cdot \Sigma_{i=1}^{n'} k_i \\ &= 3 \cdot k + 1. \end{aligned}$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If neither an error occurs nor a halting predicate is evaluated, but a cut with the initial scope is evaluated, we have

$k_{n'} + 1 \leq \ell_{n'} \leq 3 \cdot k_{n'} + 1$ instead of $k_{n'} \leq \ell_{n'} \leq 3 \cdot k_{n'} + 1$ and obtain

$$\begin{aligned} k + 1 &= k' + n + 4 + \Sigma_{i=1}^{n'} k_i \\ &\leq \ell' + n - n' + 4 + \Sigma_{i=1}^{n'} \ell_i \\ &\overset{n>0}{<} \ell' + 2 \cdot n - n' + 3 + \Sigma_{i=1}^{n'} \ell_i \\ &= \ell. \end{aligned}$$

Again, the proposition holds in this case. If neither an error occurs nor a halting predicate or a cut with the initial scope is evaluated, then the corresponding derivation is finished by two applications of the (EmptyBag) rule and the (Failure) rule. The length of the corresponding derivation is $\ell = \ell' + 2 \cdot n - n' + 4 + \Sigma_{i=1}^{n'} \ell_i$ and we have $k_i + 1 \leq \ell_i \leq 3 \cdot k_i + 1$ for all $i \in \{1, \ldots, n'\}$. Thus, we obtain

$$\begin{aligned} k &= k' + n + 3 + \Sigma_{i=1}^{n'} k_i \\ &\leq \ell' + n - n' + 2 + \Sigma_{i=1}^{n'} \ell_i \\ &< \ell' + 2 \cdot n - n' + 4 + \Sigma_{i=1}^{n'} \ell_i \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= \ell' + 2 \cdot n - n' + 4 + \Sigma_{i=1}^{n'} \ell_i \\ &\leq 3 \cdot k' + 2 \cdot n + 5 + 3 \cdot \Sigma_{i=1}^{n'} k_i \\ &< 3 \cdot k' + 3 \cdot n + 10 + 3 \cdot \Sigma_{i=1}^{n'} k_i \\ &= 3 \cdot k + 1. \end{aligned}$$

Hence, the proposition holds in this case.

– If *root*$(t)$ is the built-in predicate setof/3, then the proof is analogous to the case for bagof/3. The only difference is the order of the elements in the computed lists and that duplicates are removed.
– If *root*$(t)$ is the built-in predicate =:=/2, then we have $t = \text{'=:='}(t_1, t_2)$ and $\mathit{evaluate}(t_1 = t_2) = \mathsf{true}$. The execution continues with the goal $Q$ and the empty local substitution. Since this ISO execution takes $k' < k$ unification tests, we can use the induction hypothesis to obtain a corresponding derivation for this execution of the length $\ell'$. We construct the corresponding derivation for the initial goal by appending the obtained derivation to one application of the (ArithCompSuc) rule. Thus, we have $k = k' + 1$ and $\ell = \ell' + 1$. If the execution of $Q$ results in an error, we have $k' \leq \ell' \leq 3 \cdot k' + 1$ and the derivation also leads to an error. Then we obtain

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If the execution neither results in an error nor evaluates a halting predicate, then we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and we obtain

$$k = k' + 1 \leq \ell' < \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Thus, the proposition holds in this case.

– If $root(t)$ is the built-in predicate >/2, </2, >=/2, =</2, or =\=/2, then the proof is analogous to the case for =:=/2.
– If $root(t)$ is the built-in predicate call/1, then we have $t = \mathsf{call}(t')$ where $t'$ is a callable term. The execution continues at a new node with the query $(t'[\mathcal{V}/\mathsf{call}(\mathcal{V})], Q)$ and the empty local substitution where the scope of all cuts in $t'$ is limited to this new node. As the execution of this new node takes $k' < k$ unification tests, we can use the induction hypothesis to obtain a derivation for this execution of the length $\ell'$. We construct the corresponding derivation for the initial query by appending the obtained derivation to one application of the (Call) rule. Then the execution takes $k = k' + 1$ unification tests and the length of the derivation is $\ell = \ell' + 1$. If the execution of the second node results in an error, so does the derivation and we have $k' \leq \ell' \leq 3 \cdot k' + 1$. Thus, we obtain

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain

$$k + 1 = k' + 2 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Thus, the proposition holds in this case.
– If $root(t)$ is the built-in predicate abolish/1, !/0, ,/2, once/1, true/0, put_byte/2, put_byte/1, put_char/2, put_char/1, put_code/2, put_code/1, nl/1, nl/0, set_prolog_flag/2, close/2, close/1, set_input/1, set_output/1, open/4, open/3, flush_output/1, flush_output/0, set_stream_position/2, write_term/3, write_term/2, write/2, write/1, writeq/2, writeq/1, write_canonical/2, write_canonical/1, op/3, or char_conversion/2, then the proof is analogous to the case for call/1.
– If $root(t)$ is the built-in predicate ;/2, then we have $t = \text{';'}(t_1, t_2)$ where both $t_1$ and $t_2$ are callable terms. If $root(t_1) \neq$ ->/2, the execution continues by creating a new node with the query $(t_1, Q)$ and the empty local substitution. For the execution of the query $(t_1, Q)$ we can use the induction hypothesis as this ISO execution takes $k' < k$ unification tests. Hence, we obtain a derivation for this query of the length $\ell'$. Likewise, we obtain a derivation of the length $\ell''$ for the query $(t_2, Q)$ as the execution of this query takes $k'' < k$ unification tests if it is reached. We construct the corresponding derivation for the initial query by starting with one application of the (Disj) rule. Then we append the derivation for the query $(t_1, Q)$ where we add the query $(t_2, Q)$ before the initial scope marker to all states in that derivation

up to the first state where a cut with the initial scope is evaluated. If the execution of $(t_1, Q)$ results in an error or evaluates a halting predicate or cut with the initial scope, we are done. Then the execution takes $k = k' + 1$ unification tests and the derivation has the length $\ell = \ell' + 1$. If an error occurs, we have $k' \leq \ell' \leq 3 \cdot k' + 1$ and the derivation also results in an error. Thus, we obtain

$$k = k' + 1 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1$$

and the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, but a cut with initial scope is evaluated, we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain

$$k + 1 = k' + 2 \leq \ell' + 1 = \ell \leq 3 \cdot k' + 2 < 3 \cdot k' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If no error occurs and no halting predicate and no cut with the initial scope is evaluated, we drop the last application of the (Failure) rule from the derivation for $(t_1, Q)$ and further append the derivation for the query $(t_2, Q)$. Then the execution takes $k = k' + k'' + 1$ unification tests while the derivation has the length $\ell = \ell' + \ell'' + 1$. If the $\mathsf{ISO}$ execution of $(t_2, Q)$ results in an error, so does the derivation and we have $k'' \leq \ell'' \leq 3 \cdot k'' + 1$. Thus, we obtain

$$k = k' + k'' + 1 \leq \ell' + \ell'' < \ell' + \ell'' + 1 = \ell$$

and

$$\ell = \ell' + \ell'' + 1 \leq 3 \cdot k' + 3 \cdot k'' + 3 < 3 \cdot k' + 3 \cdot k'' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If the execution does not raise an error or evaluates a halting predicate, we have $k'' + 1 \leq \ell'' \leq 3 \cdot k'' + 1$. Thus, we obtain

$$k + 1 = k' + k'' + 2 \leq \ell' + \ell'' < \ell' + \ell'' + 1 = \ell$$

and

$$\ell = \ell' + \ell'' + 1 \leq 3 \cdot k' + 3 \cdot k'' + 3 < 3 \cdot k' + 3 \cdot k'' + 4 = 3 \cdot k + 1.$$

Again, the proposition holds in this case. If we have $t_1 = \mathsf{->}(t_3, t_4)$, then the execution continues by creating two new nodes with the queries $(\mathsf{call}(t_3), !, t_4, Q)$ and $(t_2, Q)$ and the empty local substitution. For the execution of the query $(\mathsf{call}(t_3), !, t_4, Q)$ we can use the induction hypothesis as this $\mathsf{ISO}$ execution takes $k''' < k$ unification tests. Hence, we obtain a derivation for this query with the length $\ell'''$. We construct the corresponding derivation for the initial query by starting with one application of the (IfThenElse) rule reaching the state $\langle(\mathsf{call}(t_3), !_m, t_4, Q)_{\varnothing,\varepsilon} \mid (t_2, Q)_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \; ; \; \mathcal{D} \; ; \; \mathcal{E} \; ; \; \mathcal{PI} \; ;$

$\varepsilon\rangle$. Then we append the derivation for the query $(\mathsf{call}(t_3), !, t_4, Q)$ where we replaced all $!_0$ on positions outside $Q$ by $!_m$ and then add the list of goals $(t_2, Q)_{\varnothing,\varepsilon} \mid ?_m$ before the initial scope marker (w.l.o.g. assuming that the scope $m$ is not used within the obtained derivation) to all states in that derivation up to the first state where a cut with the initial scope or the scope $m$ is evaluated. If a cut with scope $m$ was evaluated in this state we further add the goal $?_m$ before the initial scope marker to all remaining states in that derivation up to the first state where a cut with the initial scope is evaluated. If the execution of $\mathsf{call}(t_3), !, t_4, Q$ leads to an error, evaluates a halting predicate or evaluates a cut with the initial scope, we are done. Then the $\mathsf{ISO}$ execution takes $k = k''' + 1$ unification tests and the derivation has the length $\ell = \ell''' + 1$. If an error occurs, we further have $k''' \leq \ell''' \leq 3 \cdot k''' + 1$ and the derivation also results in an error. Thus, we obtain

$$k = k''' + 1 \leq \ell''' + 1 = \ell \leq 3 \cdot k''' + 2 < 3 \cdot k''' + 4 = 3 \cdot k + 1$$

and the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, but a cut with the initial scope is evaluated, we have $k''' + 1 \leq \ell''' \leq 3 \cdot k''' + 1$ and obtain

$$k + 1 = k''' + 2 \leq \ell''' + 1 = \ell \leq 3 \cdot k''' + 2 < 3 \cdot k''' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case, too. If neither an error occurs nor a halting predicate or a cut with the initial scope is evaluated, then we drop the last application of the (Failure) rule from the derivation. If a cut with the scope $m$ has been evaluated, then the constructed derivation so far ends in the state $\langle ?_m \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A\rangle$ for the dynamic part $\mathcal{D}'$ of the program, the environment $\mathcal{E}'$, the set $\mathcal{PI}'$ of user-defined predicate indicators, and list of answer substitutions $A$ reached after executing the query $(\mathsf{call}(t_3), !, t_4, Q)$. The derivation can be finished by two applications of the (Failure) rule and has, thus, the length $\ell = \ell''' + 2$ while the execution still takes $k = k''' + 1$ unification tests. Moreover, we have $k''' + 1 \leq \ell''' \leq 3 \cdot k''' + 1$. We obtain

$$k = k''' + 1 \leq \ell''' < \ell''' + 2 = \ell \leq 3 \cdot k''' + 3 < 3 \cdot k''' + 4 = 3 \cdot k + 1$$

and the proposition holds in this case. If no cut with the scope $m$ has been evaluated, then the constructed derivation so far ends in the state $\langle (t_2, Q)_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A\rangle$ and the execution takes $k = k''' + k'' + 1$ unification tests. We append the derivation for the query $(t_2, Q)$ where we added the goal $?_m$ before the initial scope marker to all states in that derivation up to the first state where a cut with the initial scope is evaluated (except the last state). If the execution of $(t_2, Q)$ leads to an error or evaluates a halting predicate or a cut with the initial scope, we are done and the derivation has the length $\ell = \ell''' + \ell''$. If an error occurs, we further have $k'' \leq \ell'' \leq 3 \cdot k'' + 1$ and the derivation also results in an error. Thus, we obtain

$$k = k''' + k'' + 1 \leq \ell''' + \ell'' = \ell$$

and

$$\ell = \ell''' + \ell'' \leq 3 \cdot k''' + 3 \cdot k'' + 2 < 3 \cdot k''' + 3 \cdot k'' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution evaluates a halting predicate, the proof is analogous. If no error occurs and no halting predicate is evaluated, but a cut with the initial scope is evaluated, we have $k'' + 1 \leq \ell'' \leq 3 \cdot k'' + 1$ and obtain

$$k + 1 = k''' + k'' + 2 \leq \ell''' + \ell'' = \ell$$

and

$$\ell = \ell''' + \ell'' \leq 3 \cdot k''' + 3 \cdot k'' + 2 < 3 \cdot k''' + 3 \cdot k'' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If neither an error occurs nor a halting predicate or a cut with the initial scope is evaluated, then we additionally change the last state of the constructed derivation so far from $\langle \varepsilon \ ; \ \mathcal{D}'' \ ; \ \mathcal{E}'' \ ; \ \mathcal{PI}'' \ ; \ A' \rangle$ to $\langle ?_0 \ ; \ \mathcal{D}'' \ ; \ \mathcal{E}'' \ ; \ \mathcal{PI}'' \ ; \ A' \rangle$ and finish the derivation with one further application of the (Failure) rule. Thus, the derivation has the length $\ell = \ell''' + \ell'' + 1$ and we have $k'' + 1 \leq \ell'' \leq 3 \cdot k'' + 1$. We obtain

$$k + 1 = k''' + k'' + 2 \leq \ell''' + \ell'' < \ell''' + \ell'' + 1 = \ell$$

and

$$\ell = \ell''' + \ell'' + 1 \leq 3 \cdot k''' + 3 \cdot k'' + 3 < 3 \cdot k''' + 3 \cdot k'' + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate ->/2, repeat/0, or not/1, then the proof is analogous to the case for ;/2.
– If $root(t)$ is the built-in predicate catch/3, then we have $t = \mathsf{catch}(g, c, r)$ where $g$ is callable. The execution continues by creating a new node with the query $(\mathsf{call}(g), Q)$ and the empty local substitution. The corresponding derivation starts with one application of the (Catch) rule reaching the state $\langle (\mathsf{call}(g))_{\varnothing,(m,c,r,Q,\varnothing)} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$. For the sub-query $(\mathsf{call}(g), Q)$ we can use the induction hypothesis to obtain a derivation of length $\ell'$ since its ISO execution takes $k' < k$ unification tests. However, in the execution of the initial query, every successful execution of $\mathsf{call}(g)$ is followed by an inactivation of the catcher $c$, which we have to count as one unification test. Let $n$ be the number of inactivations for the successful executions of $\mathsf{call}(g)$ within the execution of the initial query. If no uncaught error occurs during the execution of $(\mathsf{call}(g), Q)$, then the execution is finished and takes $k = k' + n + 1$ unification tests. The corresponding derivation is constructed

as follows. First, we add the goal $?_m$ before the initial scope marker to every state up to the first state where a cut with the initial scope is evaluated. Then every goal $(Q', Q)_{\theta,\varepsilon}$ where the evaluation of $Q$ was not reached before for this goal is replaced by $Q'_{\theta,(m,c,r,Q,\varnothing)}$. Moreover, every catch-context $(m', c', r', (Q', Q), \delta')$ is replaced by $(m', c', r', Q', \delta')$ while the catch-context $(m, c, r, Q, \varnothing)$ is added in the beginning of the list of catch-contexts in the respective goal (but only once for each goal, i.e., the replacement of several catch-contexts within one goal does not add several catch-contexts in the beginning of the respective list of catch-contexts). After this replacement, there are $n$ states in the derivation starting with the goal $\square_{\theta_i,C_i|(m,c,r,Q,\varnothing)}$ where $\theta_i$ is a substitution and $C_i$ is a list of catch-contexts for all $i \in \{1, \dots, n\}$. At each such state, an application of the (CatchNext) rule is inserted. If the execution evaluates a halting predicate, then we know $k' \leq \ell' \leq 3 \cdot k' + 1$ and the derivation is finished with the length $\ell = \ell' + n + 1$. Thus, we obtain

$$k = k' + n + 1 \leq \ell' + n + 1 = \ell$$

and

$$\ell = \ell' + n + 1 \leq 3 \cdot k' + n + 2 < 3 \cdot k' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If no halting predicate, but a cut with the initial scope is evaluated, then the construction is the same, but we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$ and obtain

$$k + 1 = k' + n + 2 \leq \ell' + n + 1 = \ell$$

and

$$\ell = \ell' + n + 1 \leq 3 \cdot k' + n + 2 < 3 \cdot k' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If no halting predicate and no cut with the initial scope is evaluated, then we finish the derivation by one further application of the (Failure) rule. Thus, the length of the derivation is $\ell = \ell' + n + 2$ and we have $k' + 1 \leq \ell' \leq 3 \cdot k' + 1$. We obtain

$$k + 1 = k' + n + 2 \leq \ell' + n + 1 < \ell' + n + 2 = \ell$$

and

$$\ell = \ell' + n + 2 \leq 3 \cdot k' + n + 2 < 3 \cdot k' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the ISO execution of $(\mathsf{call}(g), Q)$ leads to an error, we have to consider the error term $e$ which is thrown by the built-in predicate throw/1 and whether the execution of $\mathsf{call}(g)$ is still in progress. If $mgu(e', c) = \mathit{fail}$ for a fresh variant $e'$ of $e$ or the execution of $\mathsf{call}(g)$ is already achieved in the current goal, then the construction is the same as for the evaluation of a halting predicate. If the execution of $\mathsf{call}(g)$ is not yet finished, we insert one application of the (ThrowNext) rule before the final application of the (ThrowErr) rule in the constructed derivation. Both $\mathsf{ISO}$ execution and derivation lead to a program error and we have $k' \leq \ell' \leq 3 \cdot k' + 1$. If the execution of $\mathsf{call}(g)$ was already finished, then the execution takes $k = k' + n + 1$ unification tests and the derivation has the length $\ell = \ell' + n + 1$, yielding an analogous proof as for the evaluation of a halting predicate. Otherwise the execution takes $k = k' + n + 2$ unification tests and the derivation has the length $\ell = \ell' + n + 2$. Thus, we obtain

$$k = k' + n + 2 \leq \ell' + n + 2 = \ell$$

and

$$\ell = \ell' + n + 2 \leq 3 \cdot k' + n + 2 < 3 \cdot k' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If $e' \sim c$ for a fresh variant $e'$ of $e$ and the execution of $\mathsf{call}(g)$ is not yet achieved in the current goal, then let $\sigma = mgu(e', c)$. The execution continues by performing the unification test between $e'$ and $c$ instead of reaching a program error and then creates a new node with the query $(\mathsf{call}(r\sigma), Q\sigma)$ and local substitution $\sigma$. The derivation is first constructed in the same way as for the evaluation of a halting predicate, but the last application of the (ThrowErr) rule is replaced by one application of the (ThrowSuccess) rule leading to the state $\langle (\mathsf{call}(r\sigma), Q\sigma)_{\sigma,\varepsilon} \mid ?_0 \ ; \ \mathcal{D}' \ ; \ \mathcal{E}' \ ; \ \mathcal{PI}' \ ; \ A \rangle$ for the dynamic part $\mathcal{D}'$ of the program, the environment $\mathcal{E}'$, the set $\mathcal{PI}'$ of user-defined predicate indicators, and list of answer substitutions $A$ reached after the execution of $(\mathsf{call}(g), Q)$ (just before raising the error). For the query $\mathsf{call}(r\sigma), Q\sigma$ we can again use the induction hypothesis as its execution takes $k'' < k$ unification tests. Thus, we obtain a derivation of length $\ell''$ for this $\mathsf{ISO}$ execution. The corresponding derivation for the initial goal continues by appending the obtained derivation where we replaced all answer substitutions and candidate answer substitutions $\delta$ by $\sigma\delta$. If the execution of $(\mathsf{call}(r\sigma), Q\sigma)$ leads to an error or evaluates a halting predicate, then we have $k'' \leq \ell'' \leq 3 \cdot k'' + 1$ and the derivation leads to an error iff the $\mathsf{ISO}$ execution does. The execution for the initial goal takes $k = k' + k'' + n + 1$ unification tests while the corresponding derivation has the length $\ell = \ell' + \ell'' + n + 1$. Hence, we obtain

$$k = k' + k'' + n + 1 \leq \ell' + \ell'' + n + 1 = \ell$$

and

$$\ell = \ell' + \ell'' + n + 1 \leq 3 \cdot k' + 3 \cdot k'' + n + 3 < 3 \cdot k' + 3 \cdot k'' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Thus, the proposition holds in this case. If the execution of $(\mathsf{call}(r\sigma), Q\sigma)$ neither results in an error nor evaluates a halting predicate, we have $k'' + 1 \leq \ell'' \leq 3 \cdot k'' + 1$. We obtain

$$k = k' + k'' + n + 1 \leq \ell' + \ell'' + n < \ell' + \ell'' + n + 1 = \ell$$

and

$$\ell = \ell' + \ell'' + n + 1 \leq 3 \cdot k' + 3 \cdot k'' + n + 3 < 3 \cdot k' + 3 \cdot k'' + 3 \cdot n + 4 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case.

– If $\mathit{root}(t)$ is the built-in predicate $\mathsf{atom\_concat}/3$, then we have $t = \mathsf{atom\_concat}(t_1, t_2, t_3)$ where either $t_3$ is an atom and $t_1$ and $t_2$ are atoms or variables or $t_3$ is a variable and both $t_1$ and $t_2$ are atoms. The corresponding derivation starts with one application of the (AtomConcat) rule. If $t_3$ is an atom, then let $n$ be the number of characters forming $t_3$. There are $n+1$ pairs of atoms $(a_i, b_i)$ such that $t_3$ is formed by the characters of $a_i$ followed by the characters of $b_i$. The second state of the derivation is $\langle((t_1, t_2) = (a_1, b_1), Q)_{\varnothing,\varepsilon} \mid \ldots \mid ((t_1, t_2) = (a_{n+1}, b_{n+1}), Q)_{\varnothing,\varepsilon} \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon\rangle$. Let $1 \leq i \leq n+1$ be the smallest index such that $(t_1, t_2) \sim (a_i, b_i)$. If no such $i$ exists, the execution performs $k = n + 1$ failing unification tests with $n > 0$ and is finished then. The corresponding derivation continues by $n + 1$ applications of the (UnifyFail) rule followed by one application of the (Failure) rule and has, thus, the length $\ell = n + 3$. We obtain

$$k + 1 = n + 2 < n + 3 = \ell < 3 \cdot n + 4 = 3 \cdot k + 1$$

and, hence, the proposition holds in this case. If an index $i$ as described above exists, then the execution starts with $i - 1$ failing unification tests followed by one successful unification test creating a new node with the query $Q\sigma$ and local substitution $\sigma$ where $\mathit{mgu}((t_1, t_2), (a_i, b_i)) = \sigma$. For the $\mathsf{ISO}$ execution of $Q\sigma$ we can use the induction hypothesis as it takes $k_1 < k$ unification tests. Thus, we obtain a derivation of length $\ell'$ for this execution. If the execution of $Q\sigma$ results in an error or evaluates a halting predicate, so does the derivation and we have $k_1 \leq \ell' \leq 3 \cdot k_1 + 1$. We construct the derivation for the initial goal by adding the goals $((t_1, t_2) = (a_{i+1}, b_{i+1}), Q)_{\varnothing,\varepsilon} \mid \ldots \mid ((t_1, t_2) = (a_{n+1}, b_{n+1}), Q)_{\varnothing,\varepsilon}$ before the initial scope marker in the obtained derivation up to the first state where a cut with the initial scope is evaluated. Then we append this derivation to the first application of the (AtomConcat) rule. The execution takes $k = k_1 + i$ unification tests while the derivation has the length $\ell = \ell' + i + 1$. Thus, we obtain

$$k = k_1 + i \leq \ell' + i < \ell' + i + 1 = \ell$$

and

$$\ell = \ell' + i + 1 \leq 3 \cdot k_1 + i + 2 \overset{i>0}{<} 3 \cdot k_1 + 3 \cdot i + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If the execution of $Q\sigma$ neither results in an error nor evaluates a halting predicate, then we have $k_1 + 1 \leq \ell' \leq 3 \cdot k_1 + 1$. If a cut with the initial scope is evaluated, then the execution also takes $k = k_1 + i$ unification tests and the construction of the derivation for the initial goal is the same yielding a derivation of length $\ell = \ell' + i + 1$. Thus, we obtain

$$k + 1 = k_1 + i + 1 \leq \ell' + i < \ell' + i + 1 = \ell$$

and

$$\ell = \ell' + i + 1 \leq 3 \cdot k_1 + i + 2 \overset{i>0}{<} 3 \cdot k_1 + 3 \cdot i + 1 = 3 \cdot k + 1.$$

Hence, the proposition holds in this case. If no cut with the initial scope is evaluated, then the construction of the derivation is the same except that we drop the last application of the (FAILURE) rule. The execution takes $k_1 + i$ unification tests up to this point and the derivation has the length $\ell' + i$. Now the situation is similar to the beginning except that we have one child node of the initial node less for the execution and at least one goal less for the derivation. Thus, we can use the same reasoning again and this cannot be repeated more than $n+1$ times. Let $n'$ be the number of necessary unification tests for the atom_concat predicate and $n''$ be the number of successful such unification tests. Then the execution takes $k = n' + \Sigma_{j=1}^{n''} k_j$ unification tests while the derivation has the length $\ell = n' + 1 + \Sigma_{j=1}^{n''}(\ell' - 1)$. If the overall execution results in an error or evaluates a halting predicate, we have $k_j + 1 \leq \ell_j \leq 3 \cdot k_j + 1$ for all $j \in \{1, \ldots, n'' - 1\}$ and $k_{n''} \leq \ell_{n''} \leq 3 \cdot k_{n''} + 1$. We obtain

$$\begin{aligned} k &= n' + \Sigma_{j=1}^{n''} k_j \\ &\leq n' + 1 + \Sigma_{j=1}^{n''}(\ell_j - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= n' + 1 + \Sigma_{j=1}^{n''}(\ell_j - 1) \\ &\leq n' + 1 + 3 \cdot \Sigma_{j=1}^{n''} k_j \\ &< 3 \cdot n' + 1 + 3 \cdot \Sigma_{j=1}^{n''} k_j \\ &= 3 \cdot k + 1. \end{aligned}$$

Thus, the proposition holds in this case. If no error occurs and no halting predicate is evaluated, we have $k_j + 1 \leq \ell_j \leq 3 \cdot k_j + 1$ for all $j \in \{1, \ldots, n''\}$. We obtain

$$\begin{aligned} k + 1 &= n' + 1 + \Sigma_{j=1}^{n''} k_j \\ &\leq n' + 1 + \Sigma_{j=1}^{n''}(\ell_j - 1) \\ &= \ell \end{aligned}$$

and

$$\begin{aligned} \ell &= n' + 1 + \Sigma_{j=1}^{n''}(\ell_j - 1) \\ &\leq n' + 1 + 3 \cdot \Sigma_{j=1}^{n''} k_j \\ &< 3 \cdot n' + 1 + 3 \cdot \Sigma_{j=1}^{n''} k_j \\ &= 3 \cdot k + 1. \end{aligned}$$

Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate sub_atom/5, clause/2, current_predicate/1, current_flag/2 stream_property/2, current_op/3, or current_char_conversion/2, then the proof is analogous to the case for atom_concat/3.

Since the proposition holds in all cases, the first proposition is shown.

We show the second proposition by induction over the number $k$ of unification tests performed during the finite prefix of the execution w.r.t. the operational semantics of the ISO standard.

If $k = 0$, then the proposition trivially holds as for the initial query there are no answer substitutions and the initial states for our derivations also have an empty list of answer substitutions.

If $k > 0$, we can assume that the proposition holds for all finite prefixes of infinite executions with $k' < k$ unification tests. Moreover, we can already use the first proposition as its proof does not rely on the proof for the second proposition. The initial query has the form $(t, Q)$ for a term $t$ and a (possibly empty) sequence of terms $Q$. We perform a case analysis over the shape of $t$ where we already know that $root(t)$ must be a user-defined or built-in predicate, because the execution is infinite.

– If $root(t)$ is a user-defined predicate, then we have $Slice_{(\mathcal{P}|\overline{\mathcal{D}})}(t) = (c_1, \dots, c_n)$ with $n > 0$. The corresponding derivation starts with an application of the (Case) rule resulting in the state $\langle (t', Q')^{c'_1}_{\varnothing,\varepsilon} \mid \dots \mid (t', Q')^{c'_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ where $t' = t[!/!_0]$, $Q' = Q[!/!_0]$, and $c'_a = c'_a[!/!_m]$ for all $a \in \{1, \dots, n\}$. Let $i \in \{1, \dots, n\}$ be the smallest index such that the (Eval) rule is applicable to the state $\langle (t', Q')^{c'_i}_{\varnothing,\varepsilon} \mid \dots \mid (t', Q')^{c_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$. There must be such an $i$, because otherwise the execution would be finite. The execution performs $i - 1$ failing unification tests before executing the first successful one and creates a new child for the current node labeled with $(B\sigma, Q\sigma)$ and the local substitution $\sigma$ where $c_i = h :- B$ and $mgu(t, h) = \sigma$. As the $mgu$ is unique modulo variable renaming, we can w.l.o.g. assume that we have the same $mgu$ in both ISO execution and derivation. The corresponding derivation continues with $i - 1$ applications of the (Backtrack) rule followed by one application of the (Eval) rule resulting in the state $\langle (B'\sigma, Q'\sigma)_{\sigma,\varepsilon} \mid (t', Q')^{c_{i+1}}_{\varnothing,\varepsilon} \mid \dots \mid (t', Q')^{c_n}_{\varnothing,\varepsilon} \mid ?_m \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ where $c'_i = h' :- B'$. If $k \leq i$, the proposition holds as there are no answer substitutions in both the execution and derivation. Otherwise consider the execution of the query $(B\sigma, Q\sigma)$. First, we consider the case that this execution is infinite. Then we obtain a derivation with the same answer substitutions modulo variable renaming by the induction hypothesis as the last $k' = k - i$ unification tests of the prefix belong to this execution. We construct the derivation for the initial goal in the same way as in the proof for the first proposition. Thus, for each answer substitution $\delta$ in the derivation for $(B\sigma, Q\sigma)$ we now have $\sigma\delta$ as answer substitution for the initial goal. Hence, the proposition holds in this case. Now we consider the case that the execution of $(B\sigma, Q\sigma)$ is finite, i.e., it takes $k'$ unification tests. If $i + k' \geq k$, we consider the prefix of $i + k'$ unification tests. For the execution of $(B\sigma, Q\sigma)$ we obtain the corresponding derivation with the same answer substitutions modulo variable renaming by the first proposition. Let $A$ be the list of answer substitutions for the execution of $(B\sigma, Q\sigma)$. Then for the initial goal we have all answer substitutions $\sigma\delta$ such that $\delta \in A$ in the same order as in $A$. According to the construction used in the proof for the first proposition, all answer substitutions $\delta$ from the obtained derivation are replaced by $\sigma\delta$ in the corresponding derivation for the initial goal. Thus, the proposition holds in this case. If $i + k' < k$, then we obtain a derivation with the same answer substitutions modulo variable renaming for the execution of $(B\sigma, Q\sigma)$ by the first proposition. We append this derivation to the first application of the (Case) rule in the same way as in the proof for the first proposition. Now we are in a similar situation as in the beginning, but without the first child node of the initial node and with at least one goal less in the state reached by the derivation. Thus, we can apply the same construction again. This construction cannot be repeated more than $n$ times. Hence, the proposition holds in this case.

– If $root(t)$ is the built-in predicate =/2, then we have $t = {'\!='}(t_1, t_2)$ and the first unification test is performed between $t_1$ and $t_2$. Moreover, we have $t_1 \sim t_2$, because the execution takes more than one unification test. Let $mgu(t_1, t_2) = \sigma$. The ISO execution creates a new node labeled with $Q\sigma$ and $\sigma$ which is the new current node. The derivation starts with one application of the (UnifySuccess) rule reaching the state $\langle (Q\sigma)_{\sigma,\varepsilon} \mid ?_0 \ ; \ \mathcal{D} \ ; \ \mathcal{E} \ ; \ \mathcal{PI} \ ; \ \varepsilon \rangle$ (w.l.o.g. we can assume to use the same unifier in both ISO execution and derivation as an $mgu$ is unique up to variable renaming). For the execution of the query $Q\sigma$, we can use the induction hypothesis to obtain a corresponding derivation, because the execution of that query must be infinite and have a finite prefix of $k' < k$ unification tests up to the point where the prefix of $k$ unification tests for the initial query ends. We construct the derivation for the initial query by replacing every answer substitution or candidate answer substitution $\delta$ with $\sigma\delta$ in the obtained derivation, and appending this derivation to the first application of the (UnifySuccess) rule. Thus, the proposition holds in this case.

– If $root(t)$ is the built-in predicate atom_chars/2, atom_codes/2, atom_length/2, char_code/2, number_chars/2, number_codes/2, get_byte/2, get_byte/1, peek_byte/2, peek_byte/1, number_codes/2, get_char/2, get_char/1, get_code/2, get_code/1, peek_char/2, peek_char/1, peek_code/2, peek_code/1, at_end_of_stream/1, at_end_of_stream/0, current_input/1, current_output/1, arg/3, copy_term/2, functor/3, =../2, \=/2, unify_with_occurs_check/2, ==/2, \==/2, @>/2, @>=/2, @</2, @=</2, atom/1, atomic/1, compound/1, float/1, integer/1, nonvar/1, number/1, var/1, is/2, >/2, >=/2, </2, =</2, =:=/2, =\=/2, read_term/3, read_term/2, read/2, or read/1, then the proof is analogous to the case for =/2.

– If $root(t)$ is the built-in predicate call/1, then we have $t = \mathsf{call}(t')$ where $t'$ is a callable term. The execution continues at a new node with the query $(t'[\mathcal{V}/\mathsf{call}(\mathcal{V})], Q)$ and empty local substitution where the scope of all cuts in

$t'$ is limited to this new node. As the execution of this new node must be infinite and the last $k' < k$ unification tests or the finite prefix for the initial goal belong to this execution, we can use the induction hypothesis to obtain a derivation for this execution. We construct the corresponding derivation for the initial goal by appending the obtained derivation to one application of the (Call) rule. As there are no changes to the answer substitutions, the proposition holds in this case.

– If $\mathit{root}(t)$ is the built-in predicate abolish/1, asserta/1, assertz/1, !/0, ,/2, once/1, true/0, put_byte/2, put_byte/1, put_char/2, put_char/1, put_code/2, put_code/1, nl/1, nl/0, set_prolog_flag/2, close/2, close/1, set_input/1, set_output/1, open/4, open/3, flush_output/1, flush_output/0, set_stream_position/2, write_term/3, write_term/2, write/2, write/1, writeq/2, writeq/1, write_canonical/2, write_canonical/1, op/3, or char_conversion/2, then the proof is analogous to the case for call/1.
– If $\mathit{root}(t)$ is the built-in predicate retract/1, then we have $t = \mathsf{retract}(c)$. If $\mathit{root}(c) = \,:\!-/2$, then we have $c = h :\!- B$. Otherwise let $h = c$ and $B = \mathsf{true}$. Moreover, we have that $h$ is no variable, $\mathit{root}(h)$ is not a static predicate, and there is at least one clause in the program for $\mathit{root}(h)$. The first unification test is performed between $h :\!- B$ and the first clause for $\mathit{root}(h)$ in the program. Let $\mathit{Slice}_{\mathcal{D}}(t) = (h_1 :\!- B_1, \ldots, h_n :\!- B_n)$. The corresponding derivation starts with one application of the (Retract) rule. Let $i \in \{1, \ldots, n\}$ be the first index such that $h :\!- B \sim h_i :\!- B_i$ and the execution of $Q\sigma_i$ with $\mathit{mgu}(h :\!- B, h_i :\!- B_i) = \sigma_i$ is infinite (such an index must exist). Moreover, let $i_1, \ldots, i_m$ be all indices such that $i_j < i$ and $\mathit{mgu}(h :\!- B, h_j :\!- B_j) = \sigma_j$ for all $j \in \{1, \ldots, m\}$. If $k$ is smaller than the number of unification tests needed to reach the goal $Q\sigma_i$, then we consider the longer prefix up to that point. For all finite executions of the goals $Q\sigma_j$ we can use the first proposition to obtain corresponding derivations. The construction of the derivation for the initial goal is the same as the one used in the proof of the first proposition. Finally, for the goal $Q\sigma_i$ we can use the induction hypothesis and complete the construction of the derivation for the initial goal with the derivation obtained for $Q\sigma_i$. Hence, the proposition holds in this case.
– If $\mathit{root}(t)$ is the built-in predicate ->/2, repeat/0, not/1, ;/2, atom_concat/3, sub_atom/5, clause/2, current_predicate/1, current_flag/2 stream_property/2, current_op/3, or current_char_conversion/2, then the proof is analogous to the case for retract/1.
– If $\mathit{root}(t)$ is the built-in predicate findall/3, then we have $t = \mathsf{findall}(r, g, s)$ where g is callable and $s$ is a variable, partial list or a list. The execution continues with the sub-computation of the query $\mathsf{call}(g)$. If this sub-computation is infinite, we obtain a corresponding derivation by the induction hypothesis which we can use to construct a derivation for the initial query in the same way as in the proof of the first proposition. Thus, the proposition holds in this case. If the sub-computation is finite, the execution continues by unifying $s$ with the list of answer substitutions applied to $r$ (by the $\mathit{mgu}$ $\sigma$) and then executes the goal $Q\sigma$. We can use the first proposition to obtain

a corresponding derivation for the sub-computation of $\mathsf{call}(g)$. If the prefix takes less unification tests than this sub-computation and the following unification, we consider the longer prefix up to this point. By the induction hypothesis we obtain a corresponding derivation for this prefix and we construct the derivation for the initial query in the same way as in the proof of the first proposition. Hence, the proposition holds in this case.

– If $\mathit{root}(t)$ is the built-in predicate $\mathsf{bagof}/3$, then we have $t = \mathsf{bagof}(r, b, s)$ where $b = \hat{}\,(t_1, \hat{}\,(t_2, \hat{}\,(\ldots, \hat{}\,(t_m, g) \ldots)))$ for some $m \geq 0$, $\mathit{root}(g) \neq \hat{}/2$, $g$ is a callable term, and $s$ is a variable, a list, or a partial list. The execution continues with the computation of the query $\mathsf{findall}([f(X_1, \ldots, X_n), r], g, Y)$ where $\mathcal{V}(g) \setminus (\bigcup_{i=1}^{m} \mathcal{V}(t_i)) = \{X_1, \ldots, X_n\}$, $f/n \in \Sigma$ is fresh, and $Y \in \mathcal{V}$ is fresh. If this computation is infinite, we obtain the corresponding derivation by the induction hypothesis and append it to the first application of the (Bagof) rule. Thus, the proposition holds in this case. Otherwise the computation is finite and we obtain the corresponding derivation by the first proposition. The ISO execution then creates a number of child nodes of the initial node. Except for the last one, all executions of these child nodes must be finite and, hence, we obtain corresponding derivations for their executions by the first proposition. The execution of the last created child node of the initial node must be infinite. If the execution takes more than $k$ unification tests before reaching the execution of that last child node, we consider the longer prefix up to that point. By the induction hypothesis we obtain a corresponding derivation for this last child node and construct the derivation for the initial goal in the same way as in the proof of the first proposition. Hence, the proposition holds in this case.

– If $\mathit{root}(t)$ is the built-in predicate $\mathsf{setof}/3$, then the proof is analogous to the case for $\mathsf{bagof}/3$. The only difference is the order of the elements in the computed lists and that duplicates are removed.

– If $\mathit{root}(t)$ is the built-in predicate $\mathsf{catch}/3$, then we have $t = \mathsf{catch}(g, c, r)$ where $g$ is callable. The execution starts by creating a new node with the query $(\mathsf{call}(g), Q)$ and the empty local substitution. The corresponding derivation starts with one application of the (Catch) rule. If the execution of the query $(\mathsf{call}(g), Q)$ does not result in an error, then it must be infinite and we obtain a corresponding derivation by the induction hypothesis. The derivation for the initial query is constructed in the same way as in the proof of the first proposition and, thus, the second proposition holds in this case. Otherwise the execution of $(\mathsf{call}(g), Q)$ results in an error after finitely many unification tests, the catcher in the initial node is still active, and a fresh variant $e'$ of the error term $e$ being thrown unifies with $c$ by the $\mathit{mgu}$ $\sigma$. Hence, the execution continues with the query $(\mathsf{call}(r\sigma), Q\sigma)$. This execution must be infinite. If the execution takes more than $k$ unification tests up to this point, we consider the longer prefix up to this point. Therefore, we can use the first proposition to obtain a derivation for the query $(\mathsf{call}(g), Q)$ and the induction hypothesis to obtain a derivation for the query $(\mathsf{call}(r\sigma), Q\sigma)$. We construct the derivation for the initial goal in the same way as in the proof of the first proposition and the second proposition holds in this case.

Since the proposition holds in all cases, the second proposition is also shown. $\square$

## Aachener Informatik-Berichte

**This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`**

| | |
|---|---|
| 2008-01 * | Fachgruppe Informatik: Jahresbericht 2007 |
| 2008-02 | Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing |
| 2008-03 | Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination |
| 2008-04 | Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler |
| 2008-05 | Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations |
| 2008-06 | Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs |
| 2008-07 | Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition |
| 2008-08 | George B. Mertzios, Stavros D. Nikolopoulos: The $\lambda$-cluster Problem on Parameterized Interval Graphs |
| 2008-09 | George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs |
| 2008-10 | George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time |
| 2008-11 | George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows |
| 2008-12 | Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages |
| 2008-13 | Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs |
| 2008-14 | Bastian Schlich: Model Checking of Software for Microcontrollers |
| 2008-15 | Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves |
| 2008-16 | Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study |
| 2008-17 | Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving |
| 2008-18 | Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems |
| 2008-19 | Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems |

| | |
|---|---|
| 2009-02 | Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications |
| 2009-03 | Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices |
| 2009-04 | Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation |
| 2009-05 | George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs |
| 2009-06 | George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete |
| 2009-07 | Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I |
| 2009-08 | Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs |
| 2009-09 | Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem |
| 2009-10 | Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm |
| 2009-11 | Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs |
| 2009-12 | Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes |
| 2009-13 | Martin Zimmermann: Time-optimal Winning Strategies for Poset Games |
| 2009-14 | Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09) |
| 2009-15 | Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems |
| 2009-16 | George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs |
| 2009-17 | Carsten Kern: Learning Communicating and Nondeterministic Automata |
| 2009-18 | Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies |
| 2010-02 | Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time |
| 2010-03 | Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering |
| 2010-04 | René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme |
| 2010-05 | Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme |
| 2010-06 | Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata |