

Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

Jane Open Access  

Dummy University Computing Laboratory, [optional: Address], Country

My second affiliation, Country

Joan R. Public¹  

Department of Informatics, Dummy College, [optional: Address], Country

Abstract

9 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor.
10 Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem
11 ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

12 **2012 ACM Subject Classification** Replace ccsdesc macro with valid one

13 **Keywords and phrases** Dummy keyword

14 **Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

15 **Funding Jane Open Access:** (Optional) author-specific funding acknowledgements

16 **Joan R. Public:** [funding]

17 **Acknowledgements** I want to thank ...

1 Introduction

19 ELPI is a dialect of λ PROLOG (see [14, 15, 7, 12]) used as an extension language for the ROCQ
20 prover (formerly the Coq proof assistant). ELPI has become an important infrastructure
21 component: several projects and libraries depend on it [13, 3, 4, 19, 8, 9]. Examples include
22 the Hierarchy-Builder library-structuring tool [5] and Derive [17, 18, 11], a program-and-proof
23 synthesis framework with industrial applications at SkyLabs AI.

24 Starting with version 3, ELPI gained a static analysis for determinacy [10] to help users
25 tame backtracking. ROCQ users are familiar with functional programming but not necessarily
26 with logic programming and uncontrolled backtracking is a common source of inefficiency
27 and makes debugging harder. The determinacy checkers identifies predicates that behave
28 like functions, i.e., predicates that commit to their first solution and leave no *choice points*
29 (places where backtracking could resume).

30 This paper reports our first steps towards a mechanization, in the ROCQ prover, of the
31 determinacy analysis from [10]. We focus on the control operator *cut*, which is useful to
32 restrict backtracking but makes the semantic depart from a pure logical reading.

33 We formalize two operational semantics for PROLOG with cut. The first is a stack-
34 based semantics that closely models ELPI's implementation and is similar to the semantics
35 mechanized by Pusch in ISABELLE/HOL [16] and to the model of Debray and Mishra [6,
36 Sec. 4.3]. This stack-based semantics is a good starting point to study further optimizations
37 used by standard PROLOG abstract machines [20, 1], but it makes reasoning about the scope
38 of *cut* difficult. To address that limitation we introduce a tree-based semantics in which the
39 branches pruned by *cut* are explicit and we prove the two semantics equivalent. Using the

¹ Optional footnote, e.g. to mark corresponding author



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:12



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

23:2 Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

```

Inductive P := IP of nat. Inductive D := ID of nat. Inductive V := IV of nat.

Inductive Tm :=
| Tm_P of P      | Tm_D   of D      | Tm_V of V      | Tm_App of Tm & Tm.

Inductive Callable :=
| Callable_P of P | Callable_App of Callable & Tm.

```

Figure 1 Tm and Callable types

40 tree-based semantics we then show that if every rule of a predicate passes the determinacy
 41 analysis, the call to a deterministic predicate does not leave any choice points.

2 Common code: the language

put unif and progs
 gram in variables
 hides from types
 46 the them. The smallest unit of code that we can use in the langauge is an atom. The atom
 inductive (see Type 1) is either a cut or a call. A call carries a callable term (see Figure 1).
 47 A term (Tm) is either a predicate, a datum, a variable or the binary application of a term to
 another. A Callable is a term accepting predicates only predicates as functors.

```

48 Inductive A := cut | call : Callable -> A.                               (1)
49 Record R := mkR { head : Callable; premises : list A }.                  (2)
50 Record program := { rules : seq R; sig : sigT }.                         (3)
51 Definition Sigma := {fmap V -> Tm}.                                     (4)
52 Definition bc : Unif -> program -> fvS -> Callable ->
  Sigma -> fvS * seq (Sigma * R) :=                                         (5)

```

53 A rule (see Type 2) is made a head of type term and a list of premises, the premises are
 54 atoms. A program (see Type 3) is made by a list of rules and a mapping from predicates to
 55 their signatures. The type sigT is the classic type from the simply typed lambda calculus, i.e.
 56 it is either a base type or an arrow. We decorate arrows to know the mode of the lhs type.

57 A substitution (see Type 4) is a mapping from variables to terms. It is the output of a
 58 successful query and is often called the output of a query.

```

Record Unif := {
  unify : Tm -> Tm -> Sigma -> option Sigma;
  matching : Tm -> Tm -> Sigma -> option Sigma;
}.

```

59 The backchain function (bc, see Type 5) filters the rules in the program that can be
 60 used on a given query. It takes: a unificator U which explains how to unify terms up to
 61 standard unification (for output terms) or matching (for input terms); a program P to explore
 62 and filter; a set S of free variable (fvS) allowing to fresh the program P by renaming the
 63 its variables; a query q ; and the substitution σ in which the query q lives. The result of a
 64 backchain operation is couple made of an extension of S containing the new variables that
 65 have been allocated during the unification phase and a list of filtered rules r accompagnate
 66 by their a substition. This substitution is the result of the unification of q with the head of
 67 each rule in r .

68 In Figure 2, we have an example of a simple ELPI program which will be used in the
 69 following section of the paper as an example to show how backtracking and the cut operator
 70 works in the semantcis we propose. The translation of these rules in the ROCQ representation
 71 is straightforward.

```
f 1 2.   f 2 3.   r 2 4.   r 2 8.
g X X.           % r1
g X Z :- r X Z, !.    % r2
g X Z :- f X Y, f Y Z.    % r3
```

Figure 2 Small ELPI program example

2.1 The cut operator

The semantics of the cut operator adopted in the ELPI language corresponds to the *hard cut* operator of standard SWI-PROLOG. This operator has two primary purposes. First, it eliminates all alternatives that are created either simultaneously with, or after, the introduction of the cut into the execution state.

To illustrate this high-level description, consider the program shown in Figure 2 and the query $q = g 2 Z$. All three rules for g can be used on the query q . They are tried according to their order of appearance in the program: rule r_1 is tried first, followed by r_2 , and r_3 .

The first rule has no premises and immediately returns the assignment $Z = 2$. However, the computation does not terminate at this point, since two additional unexplored alternatives remain, corresponding to the premises of rules r_2 and r_3 .

The premises of rule r_2 are $r 2 Z, !$. At this stage, the role of the cut becomes apparent. If the premise $r 2 Z$ succeeds, the cut commits to this choice and removes the premises of rule r_3 from the alternative list, as they were generated at the same point as the cut. Moreover, if the call $r 2 Z$ itself produces multiple alternatives, only the first one is committed, while the remaining alternatives are discarded. This is because such alternatives have been created at a deeper depth in the search tree than the cut.

Concretely, the call $r 2 Z$ yields two solutions, assigning Z the values 4 and 8, respectively. The second solution is eliminated by the cut, and only the first assignment is preserved.

se metti $r1 = g A$
 $B :- f A B$. allora
 $g e f$ sono fun-
zioni, e puoi spie-
gare anche l'idea
del detcheck qui

3 Semantics intro

We propose two operational semantics for a logic program with cut. The two semantics are based on different syntaxes, the first syntax (called tree) exploits a tree-like structure and is ideal both to have a graphical view of its evolution while the state is being interpreted and to prove lemmas over it. The second syntax, called elpi, is the ELPI's syntax and has the advantage of reducing the computational cost of cutting and backtracking alternatives by using shared pointers. We aim to prove the equivalence of the two semantics together with some interesting lemmas of the cut behavior.

4 Tree semantics

```
Inductive tree :=
| KO | OK | TA of A
| Or  of option tree & Sigma & tree
| And of tree & seq A & tree.
```

In the tree we distinguish 5 main cases: *KO*, *OK*, and are special meta-symbols representing, respectively, the failed and a successful terminal. These symbols are considered meta because they are internal intermediate symbols used to give structure to the tree.

The *TA* constructor (acronym for tree-atom) is the constructor of atoms in the tree. TA = Todo/
Goal?

```

Fixpoint get_end s A : Sigma * tree:=
  match A with
  | TA _ | KO | OK => (s, A)
  | Or None s1 B => get_end s1 B
  | Or (Some A) _ _ => get_end s A
  | And A _ B =>
    let (s', pA) := get_end s A in
    if pA == OK then get_end s' B
    else (s', pA)
  end.

```

(a) Defintion of *get_end*

104 The two recursive cases of a tree are the *Or* and *And* non-terminals. The *Or* non-terminal
 105 $A \vee B_\sigma$ denotes a disjunction between two trees A and B . The first branch is optional, if
 106 absent it represents a dead tree, i.e. a tree that has been entirely explored. The second
 107 branch is annotated with a suspended substitution σ so that, upon backtracking to B , σ is
 108 used as the initial substitution for the execution of B .

109 The *And* non-terminal $A \wedge_{B_0} B$ represents a conjunction of two trees A and B . We call B_0
 110 the reset point for B ; it is used to restore the state of B to its initial form if a backtracking
 111 operation occurs on A . Intuitively, let $t2l$ be the function flattening a tree in a list of sequents
 112 disjunction, in PROLOG-like syntax the tree $A \wedge_{B_0} B$ becomes $(A_1, t2l B); (A_2, B_0); \dots; (A_n, B_0)$
 t2l nope, mettiti
 un r3 = 18
 X Z :- r .., if
 .., !, e rifatti
 all'esempio dell'
 sezione prima
 (fai in modo che
 f funzioni solo
 con la seconda
 regola per r)
 associate to 122
 the...
 as much as 124
 needed, indee
 prolog programs
 do not necessar 126
 ily terminate 127
 128
 129
 130
 131
 132
 133
 134

non-terminal è
 roba di gram-
 matiche, usa
 nodes/con-
 structors

A graphical representation of a tree is shown in Figure 4a. To make the graph more compact, the *And* and *Or* non-terminals are n-ary rather than binary, with right-binding priority. The *KO* terminal act as the neutral elements in the *Or* list, while *OK* is the neutral element of the *And* list.

The interpretation of a tree is performed by two main routines: *step* and *next_alt* that traverse the tree depth-first, left-to-right. Then, then *run* inductive makes the transitive closure of step *step* and *next_alt*: it iterates the calls to its auxiliary functions. In Types 7–9 we give the types contrats of these symbols where *fvS* is a set of variable names.

Inductive step_tag := Expanded | CutBrothers | Failed | Success. (6)

Definition step : program -> fvS -> Sigma -> tree -> (fvS * step_tag * tree) := (7)

Definition next_alt : bool -> tree -> option tree := (8)

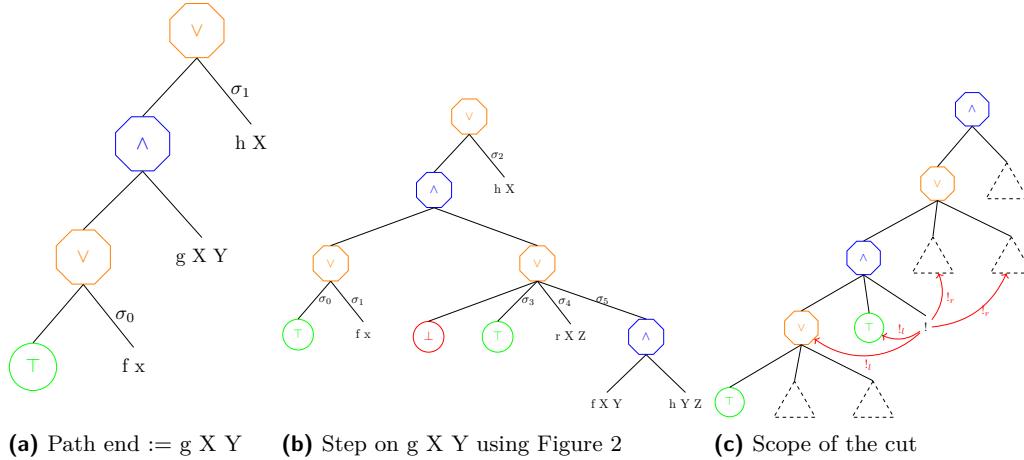
Inductive run (u:Unif) (p : program): fvS -> Sigma -> tree -> Sigma -> option tree -> Prop := (9)

The tree interpreter, as in prolog, explores the state in DFS strategy, to discover the substitution and the leaf of the tree that should be interpreted. The *get_end* routine, shown in Figure 3a, accomplishes to this task. The *get_end* returns its inputs if the tree is a leaf. Otherwise, if the tree is a disjunction, the path continues on the left subtree, if it exists, otherwise it recursively retrieves the wanted piece of information in the rhs using the substitution stored in the *Or* branch: the current substition when we cross the rhs of a *Or* is the one store in the *Or* node itself. In the case of a conjunction, if the to-be-explored leaf in the lhs is *OK*, then we look for the *get_end* in the rhs, otherwise we return the result of the lhs.

We derive the following two functions from *get_end*:

136 Definition get_subst s A := (get_end s A).1. (1)

137 Definition path_end A := (get_end empty A).2. (*empty is the empty subst*) (2)

**Figure 4** Some tree representations

138 In Figure 4a the *path_end* of the tree is `g X Y`.

139 Below we define three special kinds of trees depending on their *path_end*.

140 **Definition** `success A := path_end A == OK.` (3)

141 **Definition** `failed A := path_end A == KO.` (4)

142 **Definition** `path_atom A := if path_end A is TA _ then true else false.` (5)

143 The latter definition identifies path ending in an atom.

144 4.1 The step procedure

145 The *step* procedure takes as input a program, a set of free variables (*fv*), a substitution, and
146 a tree, and returns an updated set of free variables, a *step_tag*, and an updated tree.

147 Free variables are those variables that appear in a tree; they are used and updated when
148 a backchaining operation takes place.

149 The *step_tag* (see Type 6) indicates the kind of an internal tree step: `CutBrothers` denotes
150 the interpretation of a superficial cut, i.e., a cut whose parent nodes are all *And*-nodes.
151 `Expanded` denotes the interpretation of non-superficial cuts or predicate calls. `Failure` and
152 `Success` are returned for, respectively, `failed` and `success` trees.

153 The step procedure is intended to interpretate atoms, that is, it transforms the tree iff its
154 *path_end* is an atom, otherwsise, it returns the identity.

155 **Lemma** `succ_step_iff u p fv s A: success A <-> step u p fv s A = (fv, Success, A)`. (1)
156 **Lemma** `fail_step_iff u p fv s A: failed A <-> step u p fv s A = (fv, Failed, A)`. (2)

157 *Call step* The interpretation of a call *c* stars by calling the *bc* function on *c*. The output
158 list *l* is taken to represent build the new subtree. If *l* is empty then *KO* tree is returned,
159 otherwise the subtree is a right-skewed tree made of *n* inner *Or* nodes, where *n* is the length
160 of *l*. The root has *KO* as left child. The lhs of the other nodes is a right-skewed tree of *And*
161 nodes. The *And* nodes are again a right-seked tree containing premises of the selected rule .

162 A step in the tree of Figure 4a makes a backchain operation over the query `g X Y` and, in
163 the program defined in Figure 2, the new tree would be the one in Figure 4b. We have put a
164 red border aroung the new generated subtree. It is a disjunction of four subtrees: the first
165 node is the *KO* node (by default), the second is *OK*, since *r1* has no premises, the third and
166 the fourth contains the premises of respectively *r2* and *r3*.

dire dei reset
point

dire che le sostituzioni del backchain sono importanti e

CVIT 2016

167 *Cut step* The cut case is delicate, since interpreting a cut in a tree has three main impacts:
 168 at first the cut is replaced by the *OK* node, then some special subtrees, in the scope of the
 169 *Cut*, are cut away: in particular we need to soft-kill the left-siblings of the *Cut* and hard-kill
 170 the right-uncles of the the *Cut*.

171 ► **Definition 1** (Left-siblings (resp. right-sibling)). *Given a node A, the left-siblings (resp.*
 172 *right-sibling) of A are the list of subtrees sharing the same parent of A and that appear on*
 173 *its left (resp. right).*

174 ► **Definition 2** (Right-uncles). *Given a node A, the right-uncles of A are the list of right-sibling*
 175 *of the father of A.*

176 ► **Definition 3** (Hard-kill, $!_r$). *Given a tree t, hard-kill replaces the given subtree with the*
 177 *KO node*

178 ► **Definition 4** (Soft-kill, $!_l$). *Given a successfull tree t, soft-kill replaces with KO all subtrees*
 179 *that are not part of the path in t leading to the OK node.*

180 An example of the impact of the cut is show in Figure 4c, the dashed triangles represent
 181 generic trees. The step routine interprets the cut since it is the node in its path-end: we pass
 182 through a and and all trees on the left of the cut are successful. In the example we have 4
 183 arrow tagged with the $!_l$ or $!_r$ symbols. The $!_l$ arrows go left and soft-kill the pointed subtree,
 184 it keeps *OK* nodes since they are part of the tree leading to the cut, and replaces the other
 185 subtrees with *KO*. The $!_r$ procedure replaces the nodes pointed by the arrows with *KO*.

186 4.2 The *next_alt* procedure

187 It is evident that the *step* alone is not sufficient to reproduce entirely the behavior of the full
 188 expected prolog interpreter. In particular, we need to bracktrack on failures. Moreover, in
 189 case of success, we should return a state where the state in cleaned of the success itself, this
 190 is essential to, non deterministically, find all the solution of a given query. By Lemmas 1
 191 and 2, we know that *step* returns the identity on successful and failed states. In order to
 192 continue the computation on these particular trees, we need the *next_alt* procedure aiming
 193 to especially work with failed and successful trees: and its implementation in Figure 5.

194 The *next_alt* procedure takes a boolean and a tree, clean it from failures or success and
 195 returns a new tree if this tree still contains a non explored path. The idea behind *next_alt* is
 196 to clean recursively every subtree in DFS order if its *path_end* is a failure. Moreover, if the
 197 boolean passed to *next_alt* is true, then it erases the first successful path in the tree.

198 The base cases of *next_alt* are immidiate. The *Or* case is rathere intuitive: if the lhs
 199 of the *Or* does not exist we look for the *next_alt* in the rhs. Otherwise, we look for the
 200 *next_alt* in the lhs, if this *next_alt* does not exists, we look for the *next_alt* in the rhs.

201 We want to spend few words about the *And* case, since the reset point *B0* for *B* plays an
 202 important role. The *next_alt* in an *And* tree should consider two cases: if the lhs succeeds,
 203 then the *next_alt* should be retrived in the rhs. If this alternative does not exists it means
 204 that the rhs has entirely been explored. We need to erase the success in the lhs and try to
 205 find if a non-explored alternative exists. If so, we return a new tree with the new lhs and the
 206 rhs is built from the reset point. *big_and* is a trivial function build a right-skewed tree of
 207 and nodes where the leaves are the atoms written in the reset point. We need to reuse the
 208 reset point since, the step procedure in *And* trees evaluates the rhs of a *And* tree if the lhs
 209 succeeds. This evaluation is dependent on the subsitution in the lhs tree. Therefore, if we
 210 need to backtrack in the lhs, we need to reset the rhs.

```

Definition next_alt : bool -> tree -> option tree :=
fix next_alt b A :=
match A with
| KO => None
| OK => if b then None else Some OK
| TA _ => Some A
| And A B0 B =>
let build_B0 A := And A B0 (big_and B0) in
if success A then
  match next_alt b B with
  | None => omap build_B0 (next_alt true A)
  | Some B' => Some (And A B0 B')
  end
else if failed A then omap build_B0 (next_alt false A)
else Some (And A B0 B)
| Or None sB B => omap (fun x => Or None sB x) (next_alt b B)
| Or (Some A) sB B =>
  match next_alt b A with
  | None => omap (fun x => Or None sB x) (next_alt false B)
  | Some A' => Some (Or (Some A') sB B)
end
end.

```

■ **Figure 5** *next_alt* implementation

Some interesting property of *next_alt* are shown below and allow to see how *next_alt* complements *step*.

```

213 Lemma path_atom_next_alt_id b A: path_atom A -> next_alt b A = Some A. (3)
214 Lemma next_alt_failedF b A A': next_alt b A = Some A' -> failed A' = false. (4)

```

For example, in Figure 4b the step procedure has created a failed state: its path-end ends in *KO*. The expected behavior of *next_alt* is to take this *KO* node and make it a This allows *step* to continue the exploration of the tree. In particular, the path-end of this new state end in *OK*. The step leaves the state unchanged producing the new substitution. This solution however is not unique, we should be able to backtrack on this successful state. To do so we can call *next_alt* and it will deadify the *OK* node allowing *step* to proceed on r X Z.

subst taken form
the or

221 4.3 The *run* inductive

222 The inductive procedure *run* is modeled as a function: it takes as input a program, a set of
 223 free variables, an initial substitution σ_0 , and a tree t_0 , and returns a substitution σ_1 together
 224 with an optional updated tree t_1 . The substitution σ_1 represents the most-general unificator
 225 that makes the execution of the tree t_0 succeed starting from the initial substitution σ_0 , σ_1
 226 is an extension of σ_0 . The tree t_1 is the updated tree containing the alternatives that have
 227 not yet been explored. If the tree contains no solution, then *None* is returned.

The procedure *run* is based on three main derivation rules, shown in Figure 6. If the *path_end* of the tree *t* is a success, the input substitution is returned and the input tree is cleaned of its successful path. If the *path_end* of the tree is an atom, then *step* is invoked to evaluate this atom, and *run* is recursively called on the new tree. Finally, if the *path_end* of the tree is a failure, *next_alt* is called to clear the failed path; if the resulting cleaned tree exists, *run* is recursively called on it.

$$\begin{array}{c}
 \frac{\text{success } A \quad \text{get_subst } s1 A = s2 \quad (\text{next_alt true } A) = B}{\text{run fv } s1 A s2 B} \text{ RUN_DONE} \\
 \\
 \frac{\text{path_atom } A \quad \text{step } u p \text{ fv0 } s1 A = (\text{fv1}, \text{st}, B) \quad \text{run fv1 } s1 B s2 r}{\text{run fv0 } s1 A s2 r} \text{ RUN_STEP} \\
 \\
 \frac{\text{failed } A \quad \text{next_alt false } A = \text{Some } B \quad \text{run fv0 } s1 B s2 r}{\text{run fv0 } s1 A s2 r} \text{ RUN_FAIL}
 \end{array}$$

Figure 6 Rule system for *run***4.4 Valid trees**

235 The inductive tree allows one to generate a large number of trees, some of which are not
 236 valid, in the sense that they cannot be produced starting from a given query. The class of
 237 valid trees is characterized by the following function.

```

Fixpoint valid_tree s :=
  match s with
  | TA _ | OK | KO => true
  | Or None _ B => valid_tree B
  | Or (Some A) _ B => valid_tree A && ((B == KO) || B.base_or B)
  | And A B0 B => valid_tree A &&
    if success A then valid_tree B
    else B == big_and B0
  end.
  
```

238 Once again, the most interesting cases to analyze are *Or* and *And*.

239 For the *Or* constructor, we distinguish two cases depending on whether the left-hand
 240 side (lhs) exists. If it does not exist, then the right-hand side (rhs) must be a valid tree.
 241 Otherwise, the lhs must itself be a valid tree, and the rhs is either the *KO* tree, since it may
 242 have been removed by the evaluation of a superficial cut in the lhs, or it has not yet been
 243 explored. In the latter case, it is a *base_or* tree, namely the right-skewed tree formed by a
 244 disjunction of conjunctions.

245 For the *And* constructor, the lhs is required to be a valid tree. The shape of the rhs
 246 depends on whether the lhs represents a success. If the lhs is not successful, then the rhs
 247 has never been explored: the procedures *step* and *next_alt* modify the rhs only when the
 248 lhs succeeds. In this case, the lhs must be the right-skewed tree containing the conjunctions
 249 of the atoms present in the reset point *B*₀. In other words, the rhs coincides with the reset
 250 point. If the lhs is a success tree, then the rhs must be a valid tree.

5 Elpi semantics

252 We now want to introduce the elpi semantics. The interpreter we show reflects the interpreter
 253 of the ELPI language and is an operational semantics close to the one picked by Pusch in
 254 [16], in turn closely related to the one given by Debray and Mishra in [6, Section 4.3]. Pusch
 255 mechanized the semantics in Isabelle/HOL together with some optimizations that are present
 256 in the Warren Abstract Machine [20, 1].

257 The inductive representing a state of the ELPI language is shown below.

```

Inductive alts := no_alt | more_alt of (Sigma * goals) & alts
with goals := no_goals | more_goals of (A * alts) & goals .
  
```

```

Definition stepE fv t s a gl :=
  let (fv', rs) := bc u p fv t s in
  let rs_ca := save_alts a gl (r2a rs) in
  (fv', rs_ca).

Inductive nur : fvS -> Sigma -> goals -> alts -> Sigma -> alts -> Prop :=      (10)

  
$$\frac{}{\text{nur fv s } [:] \text{ a s a}} \text{STOPE}$$

  
$$\frac{\text{nur fv s gl ca s1 r}}{\text{nur fv s } [:] (\text{cut}, \text{ca}) \& \text{gl} \text{ a s1 r}} \text{CUTE}$$

  
$$\frac{\text{stepE fv t s al gl} = (\text{fv}', [:] \text{ b } \& \text{ bs }) \quad \text{nur fv' b.1 b.2 (bs++al) s1 r}}{\text{nur fv s } [:] (\text{call t}, \text{ca}) \& \text{gl} \text{ al s1 r}} \text{CALLE}$$

  
$$\frac{\text{stepE fv t s al gl} = (\text{fv}', [:]) \quad \text{nur fv' s1 a al s2 r}}{\text{nur fv s } [:] (\text{call t}, \text{ca}) \& \text{gl} \text{ [:] (s1, a) } \& \text{ al] s2 r.}} \text{FAILE}$$


```

Figure 7 Rule system for *nur*

258 An elpi state is an enhanced two-dimension list. The outermost list represents the list
 259 of alternatives in disjunction accompagne with the substitution that should be used to
 260 for their interpretation. The innermost list is a list of atom, representing a list of goals in
 261 conjunctions. These goals are decorated with a pointer to an elpi state, and are used to keep
 262 trace of the alternatives that should be kept when a cut is interpreted. We call these, special,
 263 alternatives the cut-to alternatives.

264 The idea of the ELPI interpreter is to receive a list of alternatives. The first alternative
 265 consists of a list of goals. Four cases must be taken into account; they are shown in Figure 7.
 266 In order to simplify goal retrieval, we split the head of the alternatives from the tail, so
 267 that it can be immediately matched in the inductive definition. Note that an empty list of
 268 alternatives represents, by definition, a failing state. If the goal list is empty (STOPE), then
 269 we have, by definition, a success, and the input solution together with the list of alternatives
 270 is returned. If the goal list starts with a cut (CUTE), then the current alternatives are erased
 271 in favour of the cut alternatives, and a recursive call is made on the remaining goal list.

272 Finally, we must consider the case in which the goal list starts with a call. The call
 273 can either succeed (CALLE) or fail (FAILE). In the latter case, a backchaining operation is
 274 mandatory: the first goal does not succeed, we need to take the head of the alternatives, and
 275 make it the new list of goals to be explored. In the former case, we need to build a new state
 276 and this ask some attention. When we are calling a rules,

277 5.1 From trees to lists

278 The translation of a tree to a list is as follows:

```

Fixpoint t2l (A: tree) s (bt : alts) : alts :=
  match A with
  | OK           => [: (s, [:]) :]
  | KO           => [:]
  | TA a         => [: (s, [: (a, [:]) :]) :]

```

```

| Or None s1 B => add_ca_deep bt (t2l B s1 [::])
| Or (Some A) s1 B =>
  let 1B := t2l B s1 [::] in
  let 1A := t2l A s 1B in
  add_ca_deep bt (1A ++ 1B)
| And A B0 B =>
  let 1A := t2l A s bt in
  if 1A is [:: (s1A, x) & xs] then
    let 1B0 := a2g B0 in
    let xz := add_deepG bt 1B0 x in
    let xs := add_deep bt 1B0 xs in
    let xs := map (catr 1B0) xs in
    let 1B := t2l B s1A (xs ++ bt) in
    (map (catl xz) 1B) ++ xs
  else [::]
end.
```

279 **6 Semantic equivalence**

► **Theorem 5** (tree_to_elpi).

280 $\forall A \sigma_1 B \sigma_2 b \sigma_0, \text{vt } A \rightarrow$
 281 $\text{run}_u \sigma_1 A (\text{Some } \sigma_2) B b \rightarrow$
 282 $\exists x xs, t2l A \sigma_1 \emptyset = x :: xs \wedge \text{nur}_u x.1 x.2 xs \sigma_2 (t2l B \sigma_0 \emptyset).$

► **Theorem 6** (elpi_to_tree).

283 $\forall \sigma_1 \sigma_2 a na g,$
 284 $\text{nur}_u \sigma_1 g a \sigma_2 na \rightarrow$
 285 $\forall \sigma_0 t, \text{vt } t \rightarrow (t2l t \sigma_0 \emptyset) = ((\sigma_1, g) :: a) \rightarrow$
 286 $\exists t' n, \text{run}_u \sigma_0 t (\text{Some } \sigma_2) t' n \wedge t2l t' \sigma_0 \emptyset = na.$

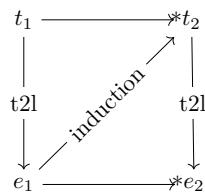
287 The proof of Theorem 6 is based on the idea explained in [2, Section 3.3]. An ideal
 288 statement for this lemma would be: given a function $12t$ transforming an elpi state to a tree,
 289 we would have that the execution of an elpi state e is the same as executing run on
 290 the tree resulting from $12t(e)$. However, it is difficult to retrieve the structure of an elpi state
 291 and create a tree from it. This is because, in an elpi state, we have no clear information
 292 about the scope of an atom inside the list and, therefore, no evident clue about where this
 293 atom should be placed in the tree.

294 Our theorem states that, starting from a valid state t which translates to a list of
 295 alternatives $(\sigma_1, g) :: a$. If we run in elpi the list of alternatives, then the execution of the
 296 tree t returns the same result as the execution in elpi. The proof is performed by induction
 297 on the derivations of the elpi execution. We have 4 derivations.

298 We have 4 cases to analyse:

299 — **References** —

- 300 1 Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 08
 301 1991. doi:10.7551/mitpress/7160.001.0001.
 302 2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488,
 303 INRIA, September 1998. URL: <https://inria.hal.science/inria-00073199v1>.

**Figure 8** Induction scheme for Theorem 6

- 304 3 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller,
 305 Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in
 306 dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve
 307 Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on*
 308 *Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages
 309 63–77. ACM, 2023. doi:10.1145/3573105.3575676.
- 310 4 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or
 311 without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages
 312 239–268, Cham, 2024. Springer Nature Switzerland.
- 313 5 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies
 314 Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPICS*, pages 34:1–34:21,
 315 2020. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPICS.FSCD.2020.34>.
 316 doi:10.4230/LIPICS.FSCD.2020.34.
- 317 6 Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J.
 318 Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.
- 319 7 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
 320 embeddable, λ Prolog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages
 321 460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/
 322 978-3-662-48899-7_32.
- 323 8 Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq
 324 Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 325 9 Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-
 326 language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.
 327 doi:10.1145/3678232.3678233.
- 328 10 Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-
 329 ming language with cut. In *Practical Aspects of Declarative Languages: 28th Interna-*
 330 *tional Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,
 331 Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.
- 332 11 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality
 333 tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing
 334 Machinery, 2023. doi:10.1145/3573105.3575683.
- 335 12 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory
 336 in higher order constraint logic programming. In *Mathematical Structures in Computer
 337 Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/
 338 S0960129518000427.
- 339 13 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least
 340 Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,
 341 *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-
 342 niz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany,
 343 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.de/entities/document/10.4230/LIPICS.ITP.2025.27](https://drops.dagstuhl.

 344 de/entities/document/10.4230/LIPICS.ITP.2025.27).
 doi:10.4230/LIPICS.ITP.2025.27.

23:12 Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

- 345 **14** Dale Miller. A logic programming language with lambda-abstraction, function variables, and
346 simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.
347 **15** Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge
348 University Press, 2012.
349 **16** Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris
350 Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors,
351 *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer
352 Berlin Heidelberg.
353 **17** Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog
354 dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January
355 2018. URL: <https://inria.hal.science/hal-01637063>.
356 **18** Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of
357 *LIPICS*, pages 29:1–29:18, September 2019. URL: <https://inria.hal.science/hal-01897468>,
358 doi:10.4230/LIPIcs.CVIT.2016.23.
359 **19** Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi.
360 Master’s thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.12568604.
361 **20** David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309,
362 SRI International, Artificial Intelligence Center, Computer Science and Technology Division,
363 Menlo Park, CA, USA, October 1983. URL: <https://www.sri.com/wp-content/uploads/2021/12/641.pdf>.