

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

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

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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

Joan R. Public: [funding]

Acknowledgements I want to thank ...

1 Introduction

Elpi is a dialect of λ Prolog (see [14, 15, 7, 12]) used as an extension language for the Rocq Prover (formerly the Coq proof assistant) that has become an important piece of infrastructure. Several projects and libraries depend on Elpi [13, 3, 4, 19, 8, 9], for example the Hierarchy-Builder library-structuring tool [5], and Derive [17, 18, 11], a program-and-proof synthesis framework with industrial applications at SkyLabs AI.

In version 3 Elpi was equipped with a static analysis for determinacy [10] to tame backtracking. Rocq users are familiar with functional programming but not necessarily with logic programming and uncontrolled backtracking is a recurrent source of inefficient and hard-to-debug code. The static analysis identifies “functions”, i.e., predicates that commit to the first result they generate by leaving no *choice points* (opportunities for backtracking).

This paper is a first step toward the mechanization in the Rocq Prover of the static analysis from [10] and it focusses on the control operator *cut*. This operator is both the ally to control backtracking and the enemy when it comes to describing the semantics of the language that becomes operational departing from the realm of logic.

This paper describes the mechanization of two operational semantics for Prolog. One operational semantic is based on a stack of choice points and reflects closely the implementation of Elpi. This semantics is close to the one mechanized by Pusch in Isabelle/HOL [16], in turn closely related to the one given by Debray and Mishra in [6, Section 4.3]. This semantics is well suited to describe some optimizations that are present in the standard Prolog abstract machine [20, 1], but is not amenable to reason about the scope of cut, that is paramount in the study of determinacy. Hence we introduce a tree-based semantics where the branches cut by the cut operator are explicit and we prove it is equivalent to the stack-based one. Finally,

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



41 using the tree-based semantics we establish that predicates where each rule passes the static
 42 analysis for determinacy do not leave choice points.

43 2 Common code: the language

```

Inductive Tm :=
| Tm_Kp      : Kp -> Tm
| Tm_Kd      : Kd -> Tm
| Tm_V       : V  -> Tm
| Tm_Comb    : Tm -> Tm -> Tm.

Inductive Callable :=
| Callable_Kp   : Kp -> Callable
| Callable_V    : V  -> Callable
| Callable_Comb : Callable -> Tm -> Callable.

Inductive RCallable :=
| RCallable_Kp   : Kp -> RCallable
| RCallable_Comb : RCallable -> Tm -> RCallable.

```

44 A callable term is a term without a data constructor as functor.

45 An rcallable is a term with rigid head.

```

Inductive A := cut | call : Callable -> A.

```

46 An atom is the smallest syntactic unit that can be executed in a prolog program \mathcal{P} .

```

Record R := mkR { head : RCallable; premises : list A }.

```

47 We exploit the typing system to ensure that the head of a "valid" rule is a term with rigid
 48 head.

```

(*simpler than in the code: signatures of preds are hidden*)
Definition program := seq R.

```

49 A program is made by a list of rules. Rules in \mathcal{P} are indexed by their position in the list.
 50 Given a list of rules \mathcal{R} and two indexes i and j , s.t. $i \neq j$ then, \mathcal{R}_i has a higher priority than
 51 \mathcal{R}_j .

52 Sigma is a substitution mapping variables to their term instantiation.

```

Definition Sigma := {fmap V -> Tm}.

```

53 The backchaining algorithm is the function \mathcal{B} aims to filter only the rules in the program
 54 \mathcal{P} having rules unifying with the current query q in a given substitution σ using the list
 55 of modes m . In particular \mathcal{B} returns for each selected rule r a substitution σ' that is the
 56 substitution obtained by the unification of the query and the head of r .

$$\mathcal{B} : (\mathcal{P}, \sigma, q) \rightarrow \text{seq}(\sigma * R)$$

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 to have a graphical view of its evaluation while the program is being interpreted. The second syntax is the elpi's syntax, we call it therefore *elpi*. We aim to prove the equivalence of the two semantics together with some interesting lemmas of the cut behavior.

3.1 Tree semantics

```

Inductive tree :=
  | Bot | OK | Dead
  | TA : A -> tree
  | Or  : tree -> Sigma -> tree -> tree
  | And : tree -> seq A -> tree -> tree.

```

In the *tree* we distinguish 6 main cases: *Bot* and *OK* are respectively the standard fail \perp and true \top predicates of prolog. *Dead* is a special symbol representing a ghost state, that is, a state useful to keep the structure of a tree from an execution to another but that is completely ignored by the interpretation of the program.

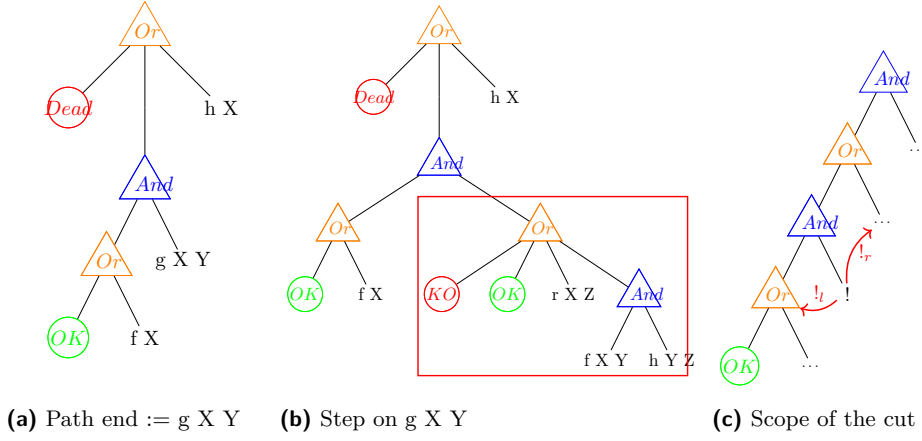
TA, standing for *tree-atom*, is a terminal of the tree containing an atom.

The two recursive cases of a tree are the *Or* and the *And* non-terminals. The *Or* non-terminals $A \vee B_\sigma$ stands for a disjunction between two trees *A* and *B*. The second tree branch is decorated with a suspended substitution σ so that, when we backtrack to *B*, we use σ as initial substitution for *B*.

The *And* non-terminal $A \wedge_{B_0} B$ represents of a conjunction of two trees *A* and *B*. We call B_0 the reset-point for *B* and is used to resume the *B* state in its initial form if some backtracking operation is performed on *A*. A graphical tree representation is shown in Figure 1a. For the sake of making our graph more compact, the *And* and *Or* non-terminals are n-ary (rather than binary), with right-binding priority. We are representing the

The interpretation of a tree is performed by two main routines: `step` and `next_alt` that traverse the tree depth-first, left-to-right.

We get the first to-be-explored terminal in the tree by getting the end of a path. This path is created from a tree traversal starting from the roots and immediately ends if the tree is not neither a disjunction, nor a conjunction: the to-be-explored terminal is the tree itself. Otherwise, if the tree is a disjunction, the path continues on the left- or the right-subtree depending of if the path of the lhs is a dead node. In the case of a conjunction, we look for the path of the lhs. If this path returns a success, we build a path in the rhs, otherwise, we return the lhs. In Figure 1a the first non-explored node is *g X*.



■ **Figure 1** Tree with first non explored node g X

87 The **step** procedure takes a tree and explores it using the path strategy. A success (i.e.
 88 a tree with path ending with OK) and failed tree (i.e. a tree with path ending with KO or
 89 Dead) is returned as it. The two interesting cases are when the path ends with a call or a
 90 cut.

91 *Call step* In the former case the call node is replaced with a new subtree made by the
 92 rules returned by the \mathcal{B} function. If \mathcal{B} returns a list l , if l is empty then KO tree is returned,
 93 otherwise the call is replaced by right-skewed tree made of n inner Or nodes, where n is
 94 the length of l . The root Or-node has KO as left child. The lhs of the other nodes is a
 95 right-skewed tree of And nodes. The And nodes are again a right-skewed tree containing then
 atoms (either cut or call) taken from the list l .

dire dei reset
point

```
g X X.                % r1
g X Z :- r X Z.       % r2
g X Z :- f X Y, h Y Z. % r3
```

97 A step in the tree in Figure 1a make a backchain operation over the query g X Y and, in
 98 the program above, the new tree would be the one in Figure 1b. We have put a red border
 99 around the new generated subtree. It is a disjunction of four subtrees: the first node is the
 100 Dead node (by default), the second is OK, since r1 has no premises, the third and the fourth
 contains the premises of respectively r2 and r3.

dire che la
sostituzioni del
backchain sono
importanti in
dove sono mess

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

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

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

111 ► **Definition 3** (Soft-kill). *Given a tree t , soft-kill replaces all the leaves of the tree with the
 112 node KO except for the leaves that are part of the path p of t .*

113 ► **Definition 4** (Hard-kill). *Given a tree t , hard-kill replaces all the leaves of the tree with the
 114 node KO*

115 An example of the impact of the cut is show in Figure 1c. The step routine interprets
 116 the cut if it is at the end of the current path. In the example we have tagged in red the
 117 arrow $!_l$ indicating which sub-trees is soft-killed and $!_r$ indicated which is sub-trees are to be
 118 hard-killed.

119 The soft-kill opeartion replace with the KO node all the

120 3.1.1 Valid tree

121 3.2 Elpi semantics

122 The Elpi interpreter is based on an operational semantics close to the one picked by Pusch
 123 in [16], in turn closely related to the one given by Debray and Mishra in [6, Section 4.3].
 124 Push mechanized the semantics in Isabelle/HOL together with some optimizations that are
 125 present in the Warren Abstract Machine [20, 1].

126 In these operational semantics we need to decorate the cut atom with a list of alternative,
 127 morally a pointer to a sub-list of the overall alternatives. An atom in the elpi semantcis is
 128 defined as follows:

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

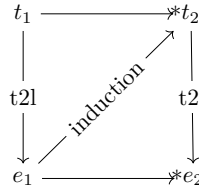
129 We are completely loosing the tree structure. There are no clean reset points. The
 130 backtracking operation is simpler: it is the tail function. The cutr and cutl operations
 131 disappears: the alternatives are stored directly in the cutE terminal.

132 The elpi interpreter is as follows:

```
(*TODO: add system of rules*)
Inductive nur : Sigma -> goals -> alts -> Sigma -> alts -> Type :=
  | StopE s a : nur s nilC a s a
  | CutE s s1 a ca r gl : nur s gl ca s1 r -> nur s ((cutE ca) ::: gl) a s1 r
  | CallE p s s1 a b bs gl r t :
    F u p t s = [:: b & bs ] ->
      nur b.1 (save_goals a gl (a2gs1 p b)) (save_alts a gl ((aa2gs p) bs) ++ a) s1 r ->
      nur s ((callE p t) ::: gl) a s1 r
  | FailE p s s1 s2 t gl a al r :
    F u p t s = [::] -> nur s1 a al s2 r -> nur s ((callE p t) ::: gl) ((s1, a) ::: al) s2 r.
```

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

```
Fixpoint t2l (A: tree) s (bt : alts) : alts :=
match A with
| OK          => [:: (s, [::])] ]
| (Bot | Dead) => [::]
| TA a        => [:: (s, [:: (a, [::])] ) ] ]
| Or A s1 B    =>
  let lB := t2l B s1 [::] in
  let lA := t2l A s lB in
  add_ca_deep bt (lA ++ lB)
```



■ **Figure 2** Induction scheme for Theorem 6

```

| And A B0 B =>
  let lB0 : goals := r2l B0 in
  let lA  := t2l A s bt in
  if lA is [:: (slA, x) & xs] then
    let xz := add_deepG bt lB0 x in
    let xs := add_deep bt lB0 xs in
    let xs := make_lB0 xs lB0 in
    let lB := t2l B slA (xs ++ bt) in
    (make_lB01 lB xz) ++ xs
  else [::]
end.

```

► **Theorem 5** (*tree_to_elpi*).

134 $\forall A \sigma_1 B \sigma_2 b \sigma_0, \forall t A \rightarrow$
 135 $\text{run}_u \sigma_1 A (\text{Some } \sigma_2) B b \rightarrow$
 136 $\exists x xs, t2l A \sigma_1 \emptyset = x :: xs \wedge \text{nur}_u x.1 x.2 xs \sigma_2 (t2l B \sigma_0 \emptyset).$

► **Theorem 6** (*elpi_to_tree*).

137 $\forall \sigma_1 \sigma_2 a na g,$
 138 $\text{nur}_u \sigma_1 g a \sigma_2 na \rightarrow$
 139 $\forall \sigma_0 t, \forall t t \rightarrow (t2l t \sigma_0 \emptyset) = ((\sigma_1, g) :: a) \rightarrow$
 140 $\exists t' n, \text{run}_u \sigma_0 t (\text{Some } \sigma_2) t' n \wedge t2l t' \sigma_0 \emptyset = na.$

141 The proof of Theorem 6 is based on the idea explained in [2, Section 3.3]. An ideal
 142 statement for this lemma would be: given a function *l2t* transforming an elpi state to a tree,
 143 we would have have that the the execution of an elpi state *e* is the same as executing *run* on
 144 the tree resulting from *l2t*(*e*). However, it is difficult to retrieve the strucutre of an elpi state
 145 and create a tree from it. This is because, in an elpi state, we have no clear information
 146 about the scope of an atom inside the list and, therefore, no evident clue about where this
 147 atom should be place in the tree.

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

152 We have 4 case to analyse:

153 — References —

- 154 1 Hassan Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 08
 155 1991. doi:10.7551/mitpress/7160.001.0001.

- 156 2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488,
157 INRIA, September 1998. URL: <https://inria.hal.science/inria-00073199v1>.
- 158 3 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller,
159 Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in
160 dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve
161 Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on*
162 *Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages
163 63–77. ACM, 2023. doi:10.1145/3573105.3575676.
- 164 4 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or
165 without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages
166 239–268, Cham, 2024. Springer Nature Switzerland.
- 167 5 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies
168 Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPICs*, pages 34:1–34:21,
169 2020. URL: [https://drops.dagstuhl.de/entities/document/10.4230/LIPICs.FSCD.2020.](https://drops.dagstuhl.de/entities/document/10.4230/LIPICs.FSCD.2020.34)
170 34, doi:10.4230/LIPICs.FSCD.2020.34.
- 171 6 Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J.*
172 *Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.
- 173 7 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
174 embeddable, λProlog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages
175 460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/
176 978-3-662-48899-7_32.
- 177 8 Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq*
178 *Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 179 9 Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-
180 language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.
181 doi:10.1145/3678232.3678233.
- 182 10 Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-
183 ming language with cut. In *Practical Aspects of Declarative Languages: 28th International*
184 *Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,
185 Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.
- 186 11 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality
187 tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing
188 Machinery, 2023. doi:10.1145/3573105.3575683.
- 189 12 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory
190 in higher order constraint logic programming. In *Mathematical Structures in Computer*
191 *Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/
192 S0960129518000427.
- 193 13 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least
194 Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,
195 *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-*
196 *niz International Proceedings in Informatics (LIPICs)*, pages 27:1–27:21, Dagstuhl, Germany,
197 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.](https://drops.dagstuhl.de/entities/document/10.4230/LIPICs.ITP.2025.27)
198 [de/entities/document/10.4230/LIPICs.ITP.2025.27](https://drops.dagstuhl.de/entities/document/10.4230/LIPICs.ITP.2025.27), doi:10.4230/LIPICs.ITP.2025.27.
- 199 14 Dale Miller. A logic programming language with lambda-abstraction, function variables, and
200 simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.
- 201 15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge
202 University Press, 2012.
- 203 16 Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris
204 Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors,
205 *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer
206 Berlin Heidelberg.

- 207 **17** Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog
208 dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January
209 2018. URL: <https://inria.hal.science/hal-01637063>.
- 210 **18** Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of
211 *LIPICs*, pages 29:1–29:18, September 2019. URL: <https://inria.hal.science/hal-01897468>,
212 doi:10.4230/LIPICs.CVIT.2016.23.
- 213 **19** Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi.
214 Master’s thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.12568604.
- 215 **20** David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309,
216 SRI International, Artificial Intelligence Center, Computer Science and Technology Division,
217 Menlo Park, CA, USA, October 1983. URL: [https://www.sri.com/wp-content/uploads/](https://www.sri.com/wp-content/uploads/2021/12/641.pdf)
218 [2021/12/641.pdf](https://www.sri.com/wp-content/uploads/2021/12/641.pdf).