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. Public[1]
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 λProlog (see [14, 15, 7, 12]) used as an extension language for the Rocq prover (formerly the Coq proof assistant). Elpi has become an important infrastructure component: several projects and libraries depend on it [13, 3, 4, 19, 8, 9]. Examples include the Hierarchy-Builder library-structuring tool [5] and Derive [17, 18, 11], a program-and-proof synthesis framework with industrial applications at SkyLabs AI.

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

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

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

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

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

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.
```

A callable term is a term without a data constructor as functor.
An rcallable is a term with rigid head.

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

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 }.
```

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

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

A program is made by a list of rules. Rules in $\mathcal{P}$ are indexed by their position in the list. 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 then $\mathcal{R}_j$.

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

Figure 1 Small program example

The elpi program above would be translated as a list of 6 elements where the heads and body are translated in the natural way.
Sigma is a substitution mapping variables to their term instantiation.

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

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

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

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 evaloution while the progrma is being intepreted. 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 $\bot$ 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 intepretation of the program.

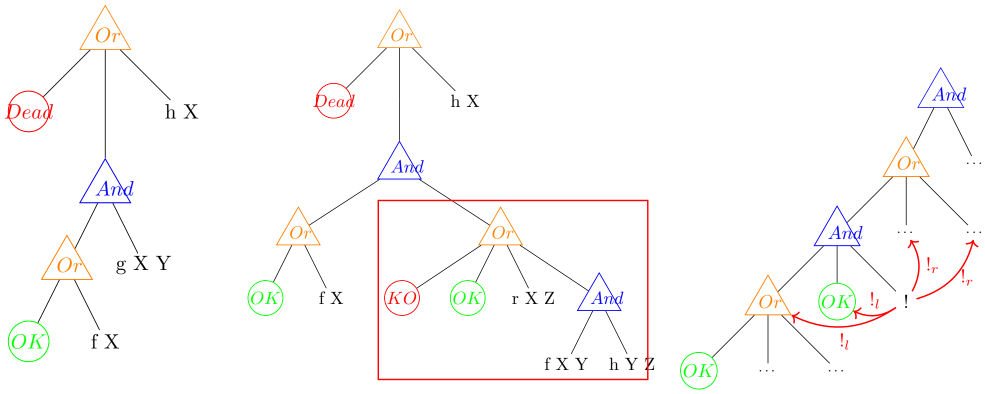
TA, standing for tree-atom, is a terminal of the tree containg an atom.

The two recursive cases of a tree are the Or and the And non-terinals. 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 substituion 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 intial form if some backtracking operation is performed on A. A graphical tree representation is shown in Figure 2a. 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 immidiatly ends if the tree is not niether 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 2a the first non-explored node is g X.



(a) Path end := g X Y **(b)** Step on g X Y using Figure 1 **(c)** Scope of the cut

Figure 2 Tree with first non explored node g X

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

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

dire dei reset point

A step in the tree in Figure 2a make a backchain operation over the query g X Y and, in the program defined in Figure 1, the new tree would be the one in Figure 2b. We have put a red border aroung the new generated subtree. It is a disjunction of four subtrees: the first node is the 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 le sostituzioni del backchain sono importanti dove sono messe

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.

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

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

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

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

An example of the impact of the cut is show in Figure 2c. The step routine interprets the cut if it is at the end of the current path. In the example we have tagged in red the arrow $!_l$ indicating which sub-trees is soft-killed and $!_r$ indicated which is sub-trees are to be hard-killed.

3.1.1 Execution example

3.1.2 Valid tree

3.2 Elpi semantics

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

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

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

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

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.
```

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)
| 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
```

$$\begin{tikzcd}
t_1 \arrow[r, "*"] \arrow[d, "\mathrm{t2l}"] & t_2 \arrow[d, "\mathrm{t2l}"] \\
e_1 \arrow[r, "*"] \arrow[ur, "\mathrm{induction}"] & e_2
\end{tikzcd}$$

Figure 3 Induction scheme for Theorem 6

```
      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).

$$\forall A\ \sigma_1\ B\ \sigma_2\ b\ \sigma_0, \boldsymbol{vt}\ A \to$$
$$\boldsymbol{run}_u\ \sigma_1\ A\ (Some\ \sigma_2)\ B\ b \to$$
$$\exists x\ xs, \boldsymbol{t2l}\ A\ \sigma_1\ \varnothing = x ::: xs \wedge \boldsymbol{nur}_u\ x.1\ x.2\ xs\ \sigma_2\ (\boldsymbol{t2l}\ B\ \sigma_0\ \varnothing).$$

▶ **Theorem 6** (elpi_to_tree).

$$\forall \sigma_1\ \sigma_2\ a\ na\ g,$$
$$\boldsymbol{nur}_u\ \sigma_1\ g\ a\ \sigma_2\ na \to$$
$$\forall \sigma_0\ t, \boldsymbol{vt}\ t \to (\boldsymbol{t2l}\ t\ \sigma_0\ \varnothing) = ((\sigma_1, g) ::: a) \to$$
$$\exists t'\ n, \boldsymbol{run}_u\ \sigma_0\ t\ (Some\ \sigma_2)\ t'\ n \wedge \boldsymbol{t2l}\ t'\ \sigma_0\ \varnothing = na.$$

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

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

We have 4 case to analyse:

References

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

2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, September 1998. URL: https://inria.hal.science/inria-00073199v1.

3 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 63–77. ACM, 2023. doi:10.1145/3573105.3575676.

4 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 239–268, Cham, 2024. Springer Nature Switzerland.

5 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPIcs*, pages 34:1–34:21, 2020. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.34, doi:10.4230/LIPIcs.FSCD.2020.34.

6 Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J. Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.

7 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages 460–468. Springer, 2015. URL: https://inria.hal.science/hal-01176856v1, doi:10.1007/978-3-662-48899-7_32.

8 Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq Workshop*, July 2023. URL: https://inria.hal.science/hal-04467855.

9 Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024. doi:10.1145/3678232.3678233.

10 Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic programming language with cut. In *Practical Aspects of Declarative Languages: 28th International Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95, Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.

11 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing Machinery, 2023. doi:10.1145/3573105.3575683.

12 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in higher order constraint logic programming. In *Mathematical Structures in Computer Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/S0960129518000427.

13 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors, *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.27, doi:10.4230/LIPIcs.ITP.2025.27.

14 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.

15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

16 Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

17 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January 2018. URL: https://inria.hal.science/hal-01637063.

18 Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of *LIPIcs*, pages 29:1–29:18, September 2019. URL: `https://inria.hal.science/hal-01897468`, `doi:10.4230/LIPIcs.CVIT.2016.23`.

19 Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi. Master's thesis, Radboud University Nijmegen, 2024. `doi:10.5281/zenodo.12568604`.

20 David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, Menlo Park, CA, USA, October 1983. URL: `https://www.sri.com/wp-content/uploads/2021/12/641.pdf`.