

# **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<sup>1</sup>**  

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  $\lambda$ 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-based  
34 semantics that closely models Elpi's implementation and is similar to the semantics mech-  
35 anized by Pusch in Isabelle/HOL [16] and to the model of Debray and Mishra [6, Sec. 4.3].  
36 This stack-based semantics is a good starting point to study further optimizations used  
37 by standard Prolog abstract machines [20, 1], but it makes reasoning about the scope of  
38 *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

---

<sup>1</sup> Optional footnote, e.g. to mark corresponding author

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

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.

### 42 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.
```

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

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

45 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 }.
```

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

```
(*simpler than in the code: signatures of preds are hidden*)  

Definition program := seq R.
```

48 A program is made by a list of rules. Rules in  $\mathcal{P}$  are indexed by their position in the list.  
 49 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  
 $\mathcal{R}_j$ .

```
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 1 Small program example

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

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

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

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

## 58 2.1 The cut operator

59 The semantics of the cut operator we have chosen in the Elpi language is the hard cut  
 60 operator used in standard SWI-Prolog. It has two main roles: it eliminates alternatives that  
 61 are chronologically created both at the same moment as, and after, the creation of the cut  
 62 operator in the execution state.

63 As a small example of this high-level definition. Let's take the program in Figure 1 and  
 64 the query  $q = g \ 2 \ Z$ . All the 3 rules for  $g$  can be used on the  $q$ . They are executed in order  
 65 of the definition in the program, i.e.,  $r1$  is tried first then  $r2$  and finally  $r3$ .

66 The first rule has no premises returns the assignment  $Z = 2$ . We however are not finished,  
 67 there are still two non-explored alternatives consisting in the premises of  $r2$  and  $r3$ .

68 The premises of  $r2$  are “ $r \ 2 \ Z, !$ ”. In this sequent the role of the cut become evident: if  
 69 it is executed, i.e.  $r \ 2 \ Z$  succeeds, then the premises of  $r3$  will be cut away, since they have  
 70 been created at the same time of the creation of the cut in the alternatives list; moreover, if  
 71 the call  $r \ 2 \ Z$  leaves alternatives, only the first is committed and the other are discarded,  
 72 since these alternatives would have a deeper depth than the cut itself.

73 Concretely speaking,  $r \ 2 \ Z$  will provide two alternatives, assigning  $Z$  respectively to 4  
 74 and 8. The second solution is discarded by the cut.

## 75 3 Semantics intro

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

### 83 3.1 Tree semantics

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

84 In the tree we distinguish 6 main cases: KO, OK, and Dead are special meta-symbols  
 85 representing, respectively, a failed, a successful, and a dead terminal. These symbols are  
 86 considered meta because they are internal intermediate symbols used to give structure to the  
 87 tree. While the first two symbols are of immediate understanding, we use Dead to represent  
 88 ghost state, that is, the Dead symbol is always ignored by the tree interpreter.

89 TA (acronym for tree-atom) is the constructor of atoms in the tree.

```

Fixpoint path_end A :=
  match A with
  | Dead | OK | KO | TA _ => A
  | Or A _ B =>
    if is_dead A then path_end B
    else path_end A
  | And A BO B =>
    match path_end A with
    | OK => path_end B
    | A => A
    end
  end.

Fixpoint is_dead A :=
  match A with
  | Dead => true
  | OK | KO | TA _ => false
  | And A BO B => is_dead A
  | Or A s B => is_dead A && is_dead B
  end.

```

(a) Defintion of `is_dead`(b) Defintion of `path_end`

90       The two recursive cases of a tree are the `Or` and `And` non-terminals. The `Or` non-terminal  
 91        $A \vee B_\sigma$  denotes a disjunction between two trees  $A$  and  $B$ . The second branch is annotated  
 92       with a suspended substitution  $\sigma$  so that, upon backtracking to  $B$ ,  $\sigma$  is used as the initial  
 93       substitution for the execution of  $B$ .

94       The `And` non-terminal  $A \wedge_{B_0} B$  represents a conjunction of two trees  $A$  and  $B$ . We call  $B_0$   
 95       the reset point for  $B$ ; it is used to restore the state of  $B$  to its initial form if a backtracking  
 96       operation occurs on  $A$ .

97       A graphical representation of the tree is shown in Figure 3a. To make the graph more  
 98       compact, the `And` and `Or` non-terminals are n-ary rather than binary, with right-binding  
 99       priority. The `KO` and `Dead` terminals act as the neutral elements in the `Or` list, while `OK` is  
 100      the neutral element of the `And` list.

101       The interpretation of a tree is performed by two main routines: `step` and `next_alt` that  
 102       traverse the tree depth-first, left-to-right. Then, then `run` inductive makes the transitive  
 103       closure of step `step` and `next_alt`: it iterates the calls to its auxiliary functions. In  
 104       Equations (2)–(4) we give the types contrats of these symbols.

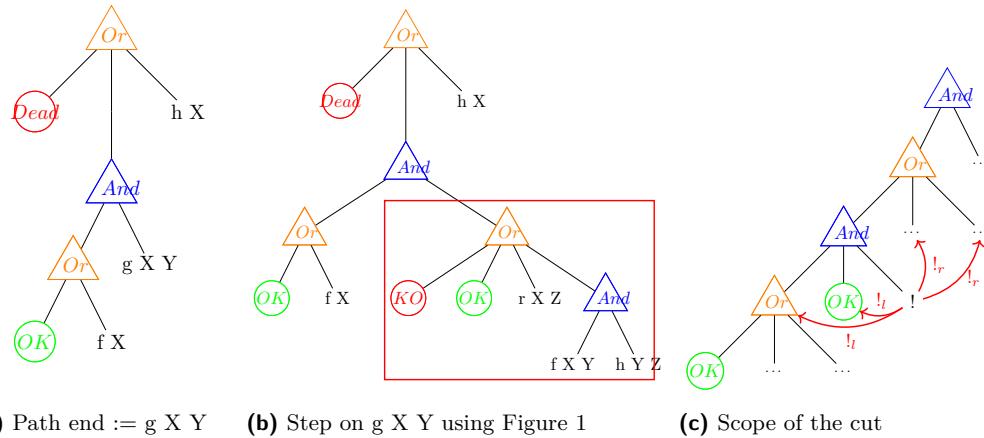
```

105 Inductive step_tag := Expanded | CutBrothers| Failure | Success. (1)
106 Definition step : program -> sigma -> tree -> (step_tag * tree) := ... (2)
107 Definition next_alt : bool -> tree -> option tree := ... (3)
108 Inductive run (p : program): Sigma -> tree -> Sigma -> tree -> bool -> Type := ... (4)

```

109       A particular tree we want to identify is a `is_dead` tree. This tree has the property  
 110       to never produce a solution: it is eiher the `Dead` tree or the lhs of `And` is dead or both  
 111       branches of `Or` are dead; `is_dead` is defined in Figure 2a. In a non-`is_dead` tree, we get the  
 112       first-to-be-explored node via `path_end`, shown in Figure 2b. The `path_end` is either the tree  
 113       itself if the tree is a leaf. Otherwise, if the tree is a disjunction, the path continues on the  
 114       left- or the right-subtree depending of if the the lhs is a `is_dead` tree, here, for example, we  
 115       see how the interpreted is ignoring a ghost state. In the case of a conjunction, it is more  
 116       interesting to see what happens. If the `path_end`  $p$  of the lhs is a success then we look for  
 117       the `path_end` in the rhs, otherwise we return  $p$ . In Figure 3a the `path_end` of the tree is  
 118       g  $X$ .

119       Below we define two special kind of trees depending on their pathend.

**Figure 3** Some tree representations

```
Definition successT A := path_end A == OK.
```

```
Definition failedT A := (path_end A == KO) || (path_end A == Dead).
```

120       The **step** procedure takes a program a substitution and a tree and returns a **step\_tag**  
 121       together with the oupdated tree. The **step\_tag** is a tag telling what kind of internal tree  
 122       step has been performed. It is either a call expansion (**Expanded**) or the evaluation of an  
 123       internal cut (i.e. a cut appering below a **Or**), a supeficial cut evaluation (**CutBrothers**), i.e. a  
 124       cut having only **And**-nodes as fathers, **Failure** or **Success** if the tree is either **successT** or  
 125       **failedT**. Stepping on a conjunction gives the Therefore, the two interesting cases of a tree  
 126       step are the step of a call and the step of a cut.  
 127

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

133       A step in the tree of Figure 3a makes a backchain operation over the query **g X Y** and, in  
 134       the program defined in Figure 1, the new tree would be the one in Figure 3b. We have put a  
 135       red border aroung the new generated subtree. It is a disjunction of four subtrees: the first  
 136       node is the KO node (by default), the second is OK, since  $r_1$  has no premises, the third and  
 137       the fourth contains the premises of respectively  $r_2$  and  $r_3$ .

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

dire dei reset  
point

dire che le  
sostituzioni del  
backchain sono  
importanti e  
dove sono mess

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

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

## 23:6 Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

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

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

151 An example of the impact of the cut is show in Figure 3c. The step routine interprets  
152 the cut if it is at the end of the current path. In the example we have tagged in red the  
153 arrow  $!_l$  indicating which sub-trees is soft-killed and  $!_r$  indicated which is sub-trees are to be  
154 hard-killed.

### 155 3.1.1 Execution example

### 156 3.1.2 Valid tree

## 157 3.2 Elpi semantics

158 TODO: dire che la semantica ad albero è più facile per le prove

159 The Elpi interpreter is based on an operational semantics close to the one picked by  
160 Pusch in [16], in turn closely related to the one given by Debray and Mishra in [6, Section  
161 4.3]. Push mechanized the semantics in Isabelle/HOL together with some optimizations that  
162 are present in the Warren Abstract Machine [20, 1].

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

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

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

169 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 .
```

170 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 | Dead) => [::]
| TA a         => [:: (s, [:: (a,[::]) ])]
| Or A s1 B   =>
  let lB := t21 B s1 [::] in
  let lA := t21 A s lB in
  add_ca_deep bt (lA ++ lB)
| And A B0 B   =>
  let lB0 : goals := r21 B0 in
  let lA := t21 A s bt in
  if lA is [:: (s1A, 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 := t21 B s1A (xs ++ bt) in
    (make_lB01 lB xz) ++ xs
  else [::]
end.

```

► **Theorem 5** (`tree_to_elpi`).

171  $\forall A \ \sigma_1 \ B \ \sigma_2 \ b \ \sigma_0, vt \ A \rightarrow$   
 172  $run_u \ \sigma_1 \ A \ (Some \ \sigma_2) \ B \ b \rightarrow$   
 173  $\exists x \ xs, t2l \ A \ \sigma_1 \ \emptyset = x :: xs \wedge nur_u \ x.1 \ x.2 \ xs \ \sigma_2 \ (t2l \ B \ \sigma_0 \ \emptyset).$

► **Theorem 6** (elpi\_to\_tree).

174  $\forall \sigma_1 \sigma_2 a na g,$   
 175  $nur_u \sigma_1 g a \sigma_2 na \rightarrow$   
 176  $\forall \sigma_0 t, vt t \rightarrow (t2l t \sigma_0 \emptyset) = ((\sigma_1, g) :: a) \rightarrow$   
 177  $\exists t' n, run_u \sigma_0 t (Some \sigma_2) t' n \wedge t2l t' \sigma_0 \emptyset = na.$

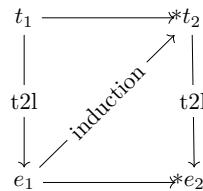
The proof of Theorem 6 is based on the idea explained in [2, Section 3.3]. An ideal statement for this lemma would be: given a function `12t` transforming an elpi state to a tree, we would have that the execution of an elpi state  $e$  is the same as executing `run` on the tree resulting from  $12t(e)$ . However, it is difficult to retrieve the structure 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 placed 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.

<sup>189</sup> We have 4 case to analyse:

190 — References

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

**Figure 4** Induction scheme for Theorem 6

- 195    3    Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller,  
 196    Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in  
 197    dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve  
 198    Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on  
 199    Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages  
 200    63–77. ACM, 2023. doi:10.1145/3573105.3575676.
- 201    4    Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or  
 202    without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages  
 203    239–268, Cham, 2024. Springer Nature Switzerland.
- 204    5    Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies  
 205    Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPICS*, pages 34:1–34:21,  
 206    2020. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.34>.  
 207    34, doi:10.4230/LIPIcs.FSCD.2020.34.
- 208    6    Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J.  
 209    Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.
- 210    7    Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,  
 211    embeddable,  $\lambda$ Prolog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages  
 212    460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/  
 213    978-3-662-48899-7\\_\\_32.
- 214    8    Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq  
 215    Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 216    9    Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-  
 217    language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.  
 218    doi:10.1145/3678232.3678233.
- 219    10    Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-  
 220    ming language with cut. In *Practical Aspects of Declarative Languages: 28th International  
 221    Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,  
 222    Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6\_5.
- 223    11    Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality  
 224    tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing  
 225    Machinery, 2023. doi:10.1145/3573105.3575683.
- 226    12    Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory in  
 227    higher order constraint logic programming. In *Mathematical Structures in Computer  
 228    Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/  
 229    S0960129518000427.
- 230    13    Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least  
 231    Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,  
 232    *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-  
 233    niz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany,  
 234    2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.27](https://drops.dagstuhl.<br/>
  235    de/entities/document/10.4230/LIPIcs.ITP.2025.27), doi:10.4230/LIPIcs.ITP.2025.27.

- 236 14 Dale Miller. A logic programming language with lambda-abstraction, function variables, and  
237 simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.
- 238 15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge  
239 University Press, 2012.
- 240 16 Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris  
241 Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors,  
242 *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer  
243 Berlin Heidelberg.
- 244 17 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi  $\lambda$ Prolog  
245 dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January  
246 2018. URL: <https://inria.hal.science/hal-01637063>.
- 247 18 Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of  
248 *LIPICS*, pages 29:1–29:18, September 2019. URL: <https://inria.hal.science/hal-01897468>,  
249 doi:10.4230/LIPIcs.CVIT.2016.23.
- 250 19 Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi.  
251 Master’s thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.12568604.
- 252 20 David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309,  
253 SRI International, Artificial Intelligence Center, Computer Science and Technology Division,  
254 Menlo Park, CA, USA, October 1983. URL: <https://www.sri.com/wp-content/uploads/2021/12/641.pdf>.