

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-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

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. 23:1–23:9
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.

42 2 Common code: the language

43 Before going to the two semanticis, we show the piece of data structure that are shared by
44 the them. The smallest unit of code that we can use in the langauge is an atom. The atom
45 inductive (see Type 1) is either a cut or a call. A call carries a callable term (see Figure 1).
46 A term (Tm) is either a predicate, a datum, a variable or the binary application of a term to
47 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(5) R)) :=
```

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 variales 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 Below we have an example of a simple elpi program which will be used in the following
69 section of the paper as an example to show how backtracking and the cut operator works
70 in the semantcis we propose. The translation of these rules in the Rocq representation is
71 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

72 2.1 The cut operator

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

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

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

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

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

89 3 Semantics intro

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

97 3.1 Tree semantics

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

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

103 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*

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 second branch is annotated
 106 with a suspended substitution σ so that, upon backtracking to B , σ is used as the initial
 107 substitution for the execution of B .

108 The *And* non-terminal $A \wedge_{B_0} B$ represents a conjunction of two trees A and B . We
 109 call B_0 the reset point for B ; it is used to restore the state of B to its initial form if a
 110 backtracking operation occurs on A . Intuitively in prolog-like syntax, in a tree $A \wedge_{B_0} B$, if
 111 $t2l$ is the function flattening the tree in a list of sequents disjnction and $t2l(A) = A_1, \dots, A_n$,
 112 then we would have $(A_1, t2l B); (A_2, B0); \dots; (A_n, B0)$.

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

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

```

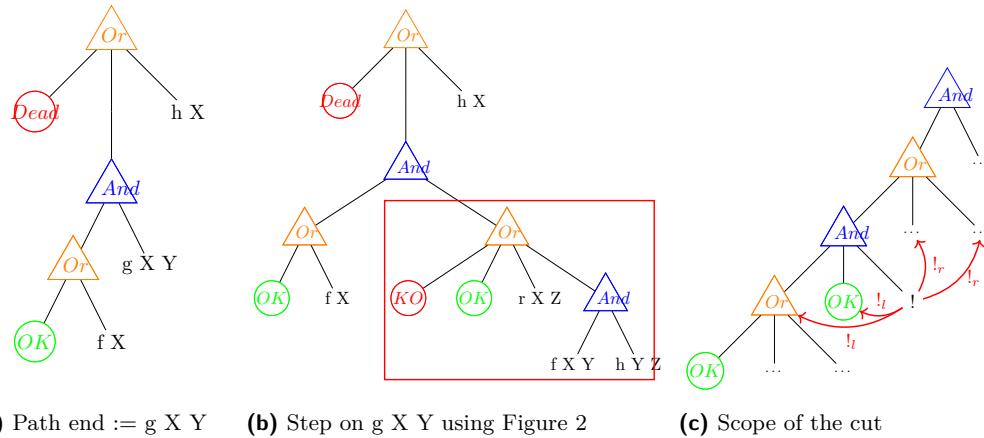
121 Inductive step_tag := Expanded | CutBrothers | Failed | Success.           (6)
122 Definition step : program -> fvS -> Sigma -> tree -> (fvS * step_tag * tree) := (7)
123 Definition next_alt : bool -> tree -> option tree :=                      (8)
124 Inductive run (p : program) : fvS -> Sigma -> tree -> option Sigma -> tree -> bool -> Prop := (9)

```

125 A particular tree we want to identify is a *is_dead* tree (defined in Figure 3a). This tree
 126 has the property to never produce a solution: it is eiher the *Dead* tree or both branches of
 127 *Or* are dead, or the lhs of *And* is dead. In the latter case, we note that B can be non-dead,
 128 but this is not a problem since the interpreter can run B only if A is non-dead.

129 The prolog interpreter explores the state in DFS strategy, it finds the “first-to-be-explored”
 130 (ftbe) atom of the tree and then interpretes it. In a non-*is_dead* tree, we get the ftbe node
 131 via *path_end*, shown in Figure 3b. The *path_end* is either the tree itself if the tree is a leaf.
 132 Otherwise, if the tree is a disjunction, the path continues on the left- or the right-subtree
 133 depending of if the the lhs is a *is_dead* tree. In the *Or* case we are clearing ignoring the
 134 dead (ghost) state.

135 In the case of a conjunction, it is more interesting to see what happens. If the *path_end*
 136 p of the lhs is a success then we look for the *path_end* in the rhs, otherwise we return p . In

**Figure 4** Some tree representations

137 Figure 4a the *path_end* of the tree is *g X*.

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

139 **Definition** *success* *A* := *path_end* *A* == *OK*. (1)

140 **Definition** *failed* *A* := (*path_end* *A* == *KO*) || (*path_end* *A* == *Dead*). (2)

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

142 Free variables are those variables that appear in a tree; they are used in the backchaining operation to refresh the variables in the program.

143 The *step_tag* indicates the type of internal tree step that has been performed. *CutBrothers* denotes the interpretation of a superficial cut, i.e., a cut whose parent nodes are all *And*-nodes. *Expanded* denotes the interpretation of non-superficial cuts or predicate calls. *Failure* and *Success* are returned for, respectively, *failed* and *success* trees.

144 The step procedure is intended to interpretate atoms, that is, it returns the identity for *success* and *failed* tree.

145 **Lemma** *success_step* *u p fv s A*: *success A* \rightarrow *step u p fv s A* = (*fv, Success, A*). (1)
 146 **Lemma** *failed_step* *u p fv s1 A*: *failed A* \rightarrow *step u p fv s1 A* = (*fv, Failed, A*). (2)

147 Therefore, the two interesting cases of a tree the interpretation of trees with path-end equal to a call or a cut atom.

148 *Call step* The interpretation of a call *c* is performed by replacing the call wrt the result of the *B c*, then 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 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*.

149 A step in the tree of Figure 4a makes a backchain operation over the query *g X Y* and, in 150 the program defined in Figure 2, the new tree would be the one in Figure 4b. We have put a 151 red border around the new generated subtree. It is a disjunction of four subtrees: the first 152 node is the *KO* node (by default), the second is *OK*, since *r1* has no premises, the third and 153 the fourth contains the premises of respectively *r2* and *r3*.

154 *Cut step* The latter case is delicate since interpreting a cut in a tree has three main 155 impacts: at first it is replaced by the *OK* node, then some special subtrees, in the scope 156 of the *Cut*, are cut away: in particular we need to soft-kill the left-siblings of the *Cut* and 157 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

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

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

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

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

178 An example of the impact of the cut is show in Figure 4c. The step routine interprets
 179 the cut if it is at the end of the current path. In the example we have tagged in red the
 180 arrow $!_l$ indicating which sub-trees is soft-killed and $!_r$ indicated which is sub-trees are to be
 181 hard-killed.

182 3.1.1 Execution example

183 3.1.2 Valid tree

184 3.2 Elpi semantics

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

186 The Elpi interpreter is based on an operational semantics close to the one picked by
 187 Pusch in [16], in turn closely related to the one given by Debray and Mishra in [6, Section
 188 4.3]. Push mechanized the semantics in Isabelle/HOL together with some optimizations that
 189 are present in the Warren Abstract Machine [20, 1].

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

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

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

196 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.
```

197 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 1B := t2l B s1 [::] in
  let 1A := t2l A s 1B in
  add_ca_deep bt (1A ++ 1B)
| And A B0 B  =>
  let 1B0 : goals := r2l B0 in
  let 1A := t2l A s bt in
  if 1A is [:: (s1A, x) & xs] then
    let xz := add_deepG bt 1B0 x in
    let xs := add_deep bt 1B0 xs in
    let xs := make_1B0 xs 1B0 in
    let 1B := t2l B s1A (xs ++ bt) in
    (make_1B01 1B xz) ++ xs
  else [::]
end.
```

► **Theorem 5** (tree_to_elpi).

198 $\forall A \sigma_1 B \sigma_2 b \sigma_0, vt A \rightarrow$
 199 $run_u \sigma_1 A (Some \sigma_2) B b \rightarrow$
 200 $\exists x xs, t2l A \sigma_1 \emptyset = x ::: xs \wedge nur_u x.1 x.2 xs \sigma_2 (t2l B \sigma_0 \emptyset).$

► **Theorem 6** (elpi_to_tree).

201 $\forall \sigma_1 \sigma_2 a na g,$
 202 $nur_u \sigma_1 g a \sigma_2 na \rightarrow$
 203 $\forall \sigma_0 t, vt t \rightarrow (t2l t \sigma_0 \emptyset) = ((\sigma_1, g) :: a) \rightarrow$
 204 $\exists t' n, run_u \sigma_0 t (Some \sigma_2) t' n \wedge t2l t' \sigma_0 \emptyset = na.$

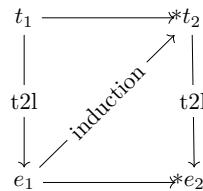
205 The proof of Theorem 6 is based on the idea explained in [2, Section 3.3]. An ideal
 206 statement for this lemma would be: given a function $12t$ transforming an elpi state to a tree,
 207 we would have have that the the execution of an elpi state e is the same as executing run on
 208 the tree resulting from $12t(e)$. However, it is difficult to retrive the strucutre of an elpi state
 209 and create a tree from it. This is because, in an elpi state, we have no clear information
 210 about the scope of an atom inside the list and, therefore, no evident clue about where this
 211 atom should be place in the tree.

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

216 We have 4 case to analyse:

217 ————— **References** —————

- 218 1 Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 08
 219 1991. doi:10.7551/mitpress/7160.001.0001.

**Figure 5** Induction scheme for Theorem 6

- 220 2 Yves Bertot. A certified compiler for an imperative language. Technical Report RR-3488,
221 INRIA, September 1998. URL: <https://inria.hal.science/inria-00073199v1>.
- 222 3 Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller,
223 Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in
224 dependent type theory. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve
225 Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on
226 Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages
227 63–77. ACM, 2023. doi:10.1145/3573105.3575676.
- 228 4 Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or
229 without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages
230 239–268, Cham, 2024. Springer Nature Switzerland.
- 231 5 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies
232 Made Easy in Coq with Elpi. In *Proceedings of FSCD*, volume 167 of *LIPICS*, pages 34:1–34:21,
233 2020. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPICS.FSCD.2020.34>.
234 doi:10.4230/LIPICS.FSCD.2020.34.
- 235 6 Saumya K. Debray and Prateek Mishra. Denotational and operational semantics for prolog. *J.
236 Log. Program.*, 5(1):61–91, March 1988. doi:10.1016/0743-1066(88)90007-6.
- 237 7 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
238 embeddable, λ Prolog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages
239 460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/
240 978-3-662-48899-7_32.
- 241 8 Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq
242 Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 243 9 Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-
244 language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.
245 doi:10.1145/3678232.3678233.
- 246 10 Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-
247 ming language with cut. In *Practical Aspects of Declarative Languages: 28th International
248 Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,
249 Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.
- 250 11 Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality
251 tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing
252 Machinery, 2023. doi:10.1145/3573105.3575683.
- 253 12 Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory
254 in higher order constraint logic programming. In *Mathematical Structures in Computer
255 Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/
256 S0960129518000427.
- 257 13 Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least
258 Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,
259 *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-
260 niz International Proceedings in Informatics (LIPICS)*, pages 27:1–27:21, Dagstuhl, Germany,

- 261 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.
- 262 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.
- 263 15 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.
- 264 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.
- 265 17 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January 2018. URL: <https://inria.hal.science/hal-01637063>.
- 266 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.
- 267 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.
- 268 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>.