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

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

Inductive Tm :=                            Inductive Callable :=
  | Tm_P of P     | Tm_D   of D              | Callable_P   of P
  | Tm_V of V     | Tm_App of Tm & Tm.       | Callable_App of Callable & Tm.
```

Figure 1 Tm and Callable types

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

2 Common code: the language

put unif and program in variable hides from types

Before going to the two semantcis, we show the piece of data structure that are shared by the them. The smallest unit of code that we can use in the langauge is an atom. The atom inductive (see Type 1) is either a cut or a call. A call carries a callable term (see Figure 1). A term (Tm) is either a predicate, a datum, a variable or the binary application of a term to another. A Callable is a term accepting predicates only predicates as functors.

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

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

A substitution (see Type 4) is a mapping from variables to terms. It is the output of a 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;
}.
```

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

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

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

Figure 2 Small ELPI program example

2.1 The cut operator

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

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

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

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

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

3 Semantics intro

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

3.1 Tree semantics

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

In the tree we distinguish 6 main cases: *KO*, *OK*, and *Dead* are special meta-symbols representing, respectively, a failed, a successful, and a dead terminal. These symbols are considered meta because they are internal intermediate symbols used to give structure to the

```
Fixpoint path_end A :=
  match A with
  | OK | KO | TA _ => A
  | Or None _ B => path_end B
  | Or (Some A) _ _ => path_end A
  | And A _ B =>
    match path_end A with
    | OK => path_end B
    | A => A
    end
  end.
```

(a) Defintion of *path_end*

tree. While the first two symbols are of immiediate understanding, we use *Dead* to represent ghost state, that is, the *Dead* symbol is always ignored by the tree interpreter.

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

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

The *And* non-terminal $A \wedge_{B_0} B$ represents a conjunction of two trees A and B. We call B_0 the reset point for B; it is used to restore the state of B to its initial form if a backtracking operation occurs on A. Intuitively, let *t2l* be the function flattening a tree in a list of sequents disjnction, in PROLOG-like syntax the tree $A \wedge_{B_0} B$ becomes $(A_1, \mathit{t2l}\, B); (A_2, B_0); \ldots; (A_n, B_0)$ where $\mathit{t2l}(A) = A_1, \ldots, A_n$.

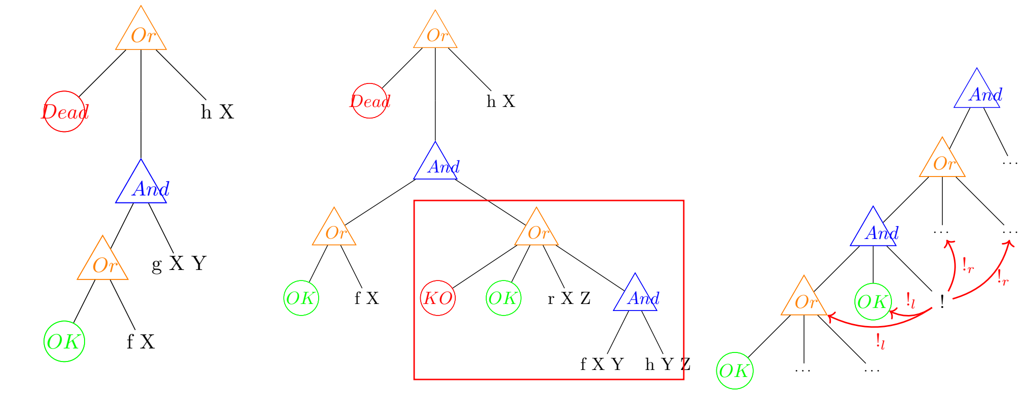
A graphical representation of a tree is shown in Figure 4a. To make the graph more compact, the *And* and *Or* non-terminals are n-ary rather than binary, with right-binding priority. The *KO* and *Dead* terminals act as the neutral elements in the *Or* list, while *OK* is the neutral element of the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

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

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

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

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



(a) Path end := g X Y **(b)** Step on g X Y using Figure 2 **(c)** Scope of the cut

Figure 4 Some tree representations

In the case of a conjunction, it is more interesting to see what happens. If the *path_end* p of the lhs is a success then we look for the *path_end* in the rhs, otherwise we return p. In Figure 4a the *path_end* of the tree is `g X`.

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

```
Definition success A := path_end A == OK.          (1)
Definition failed A := (path_end A == KO).         (2)
```

3.1.1 The *step* procedure

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.

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

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.

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

```
Lemma success_step u p fv s A: success A -> step u p fv s A = (fv, Success, A). (1)
Lemma failed_step u p fv s1 A: failed A -> step u p fv  s1 A = (fv, Failed, A). (2)
```

Therefore, *step* produces interesting results if the path-end of the input tree is either a call or a cut.

Call step The interpretation of a call c stars by calling the bc function on c. The output list l is taken to represent build the new subtree. If l is empty then *KO* tree is returned, otherwise the subtree is a 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 premises of the selected rule .

if we go right in the tree, the subst is the one in the or... dire dei reset point

A step in the tree of Figure 4a makes a backchain operation over the query `g X Y` and, in the program defined in Figure 2, the new tree would be the one in Figure 4b. We have put a red border aroung the new generated subtree. It is a disjunction of four subtrees: the first node is the *KO* node (by default), the second is *OK*, since r1 has no premises, the third and dire che le sostituzioni del backchain sono importanti e dove sono mess

the fourth contains the premises of respectively r_2 and r_3.

Cut step The latter case is delicate since interpreting a cut in a tree has three main impacts: at first it is replaced by the *OK* node, then some special subtrees, in the scope of the *Cut*, are cut away: 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, $!_l$). *Given a successfull tree t, soft-kill replaces all the leaves of the tree with the node* KO *except for the path in t leading to the* OK *node.*

▶ **Definition 4** (Hard-kill, $!_r$). *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 4c. The step routine interprets the cut since it is the node in its path-end. In the example we have 4 arrow tagged with the $!_l$ or $!_r$ symbols. The $!_l$ arrows go left and soft-kill the pointed subtree, in particular, we can note that both pointed subtree have a success node, this is beacuse, in order to evaluate the cut in the figure, we need a successful path leading to it. The $!_l$ procedure will keep the two *OK* nodes since they are essential to reach the cut, and will kill all the leaves in the other subtrees, for those specific subtrees, $!_l$ behaves as $!_r$. The $!_r$ procedure, instead, immediately starts by removing all leaves in the trees pointed by the red arrows.

dire che step non aggiunge mai nuovi dead

3.1.2 The *next_alt* procedure

It is evident that the *step* alone is not sufficient to reproduce entirely the behavior of the full ELPI solver. In particular, *step* does not perform any backtracking at all: it does not backtrack neither for failures, nor for success, from Lemmas 1 and 2, *step* returns the identity. To do so, we have the *next_alt* procedure: its signature is provided in Type 8 and its implementation in Figure 5.

The *next_alt* procedure takes a boolean and a tree and return a new tree if it still contains an alternative. The intuition of *next_alt* is to introduce trasnform failed (or success) path into dead-path by inserting new Dead nodes. The boolean tells if there success leaves should be

that is it is allowed to transform *OK* or *KO* leaves into *Dead*, so that the *step* procedure is allowed to ignore the new ghosts states and move on. The boolean taken by *next_alt* tells if it is needed to kill *OK* nodes or not.

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

subst taken form the or

More concretely the code for *next_alt* is show in

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

Figure 5 *next_alt* implementation

3.1.3 The *run* inductive

3.1.4 Valid tree

Reasoning on a the tree semantics allows to identify an invariant that

3.2 Elpi semantics

TODO: dire che la semantica ad albero è pużù faicle per le prove

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, [::]) ]
| KO          => [::]
| TA a        => [:: (s, [:: (a,[::]) ]) ]
| Or A s1 B   =>
    let lB := t2l B s1 [::] in
    let lA := if A is Some A then t2l A s lB else [::] in
    add_ca_deep bt (lA ++ lB)
| And A B0 B   =>
    let lB0 : goals := r2l B0 in
    let lA  := t2l A s bt in
```

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

Figure 6 Induction scheme for Theorem 6

```
    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 := map (catr lB0) xs in
      let lB := t2l B slA (xs ++ bt) in
      (map (catl xz) lB) ++ xs
    else [::]
end.
```

▶ **Theorem 5** (`tree_to_elpi`).

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

▶ **Theorem 6** (`elpi_to_tree`).

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

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`.