

Operational semantics for Prolog with Cut in Rocq and its application to determinacy analysis

Jane Open Access   

Dummy University Computing Laboratory, [optional: Address], Country

My second affiliation, Country

Joan R. Public¹  

Department of Informatics, Dummy College, [optional: Address], Country

Abstract

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

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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

Joan R. Public: [funding]

Acknowledgements I want to thank ...

1 Introduction

ELPI is a dialect of λ PROLOG (see [14, 15, 7, 12]) used as an extension language for the ROCQ prover (formerly the COQ proof assistant). 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



© 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. 23:1–23:11

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```

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

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 pro
gram in variable
hides from types

Before going to the two semantics, we show the piece of data structure that are shared by the them. The smallest unit of code that we can use in the language 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 unificator U which explains how to unify terms up to standard unification (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 substitution. 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 semantics we propose. The translation of these rules in the ROCQ representation is 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

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 interpreted 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 of A
  | Or of option tree & Sigma & tree
  | And of tree & seq A & tree.

```

In the tree we distinguish 5 main cases: KO , OK , and are special meta-symbols representing, respectively, the failed and a successful terminal. These symbols are considered meta because they are internal intermediate symbols used to give structure to the tree.

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

se metti $r1 = g\ A$
 $B :- f\ A\ B$. allora
 $g\ e\ f$ sono funz-
 zioni, e puoi spie-
 gare anche l'idea
 del detach qui

$TA =$ Todo/-
 Goal?

```

Fixpoint get_end s A : Sigma * tree :=
  match A with
  | TA _ | KO | OK => (s, A)
  | Or None s1 B => get_end s1 B
  | Or (Some A) _ => get_end s A
  | And A _ B =>
    let (s', pA) := get_end s A in
    if pA == OK then get_end s' B
    else (s', pA)
  end.

```

(a) Definition of *get_end*

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 first branch is optional, if absent it represents a dead tree, i.e. a tree that has been entirely explored. 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 disjunction, in PROLOG-like syntax the tree $A \wedge_{B_0} B$ becomes $(A_1, t2l B); (A_2, B_0); \dots; (A_n, B_0)$ where $t2l(A) = A_1, \dots, A_n$.

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* terminal 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* and *next_alt*: it iterates the calls to its auxiliary functions. In Types 7–9 we give the types contrats of these symbols where *fvS* 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 (u:Unif) (p : program): fvS -> Sigma -> tree -> Sigma -> option tree -> Prop := (9)

The tree interpreter, as in prolog, explores the state in DFS strategy, to discover the substitution and the leaf of the tree that should be interpreted. The *get_end* routine, shown in Figure 3a, accomplishes to this task. The *get_end* returns its inputs if the tree is a leaf. Otherwise, if the tree is a disjunction, the path continues on the left subtree, if it exists, otherwise it recursively retrieves the wanted piece of information in the rhs using the substitution stored in the *Or* branch: the current substitution when we cross the rhs of a *Or* is the one store in the *Or* node itself. In the case of a conjunction, if the to-be-explored leaf in the lhs is *OK*, then we look for the *get_end* in the rhs, otherwise we return the result of the lhs.

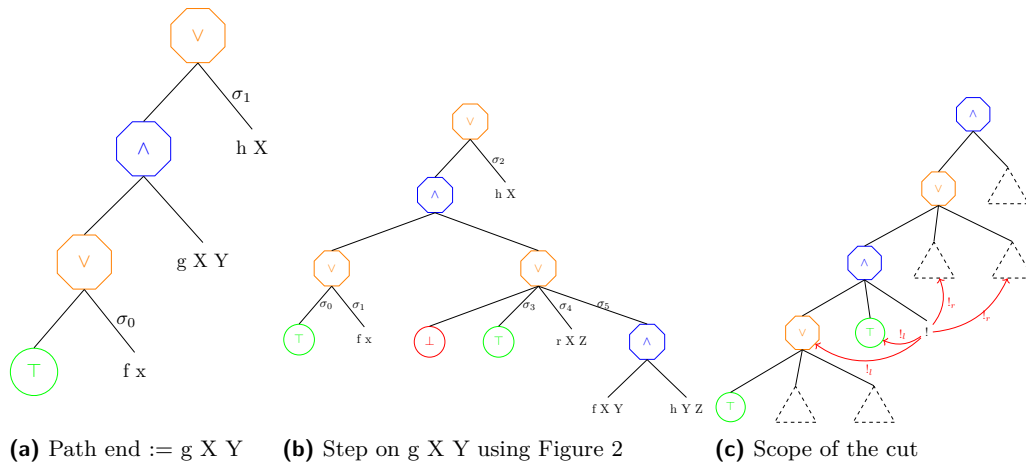
We derive the following two functions from *get_end*:

Definition get_subst s A := (get_end s A).1. (1)

Definition path_end A := (get_end empty A).2. (*empty is the empty subst*) (2)

non-terminal è
roba di gram-
matiche, usa
nodes/con-
structors

t2l nope, mett
un r3 = 18
X Z :- r .., 116
.., !, e rifatti
all'esempio della
sezione prima
(fai in modo che
f funzioni solo
con la seconda
regola per r)
associate to
the...
as much as
needed, indee
prolog programs
do not necessar
ily terminate



■ **Figure 4** Some tree representations

138 In Figure 4a the *path_end* of the tree is *g X Y*.

139 Below we define three special kinds of trees depending on their *path_end*.

140 **Definition** *success* A := *path_end* A == OK. (3)

141 **Definition** *failed* A := *path_end* A == KO. (4)

142 **Definition** *path_atom* A := if *path_end* A is TA _ then true else false. (5)

143 The latter definition identifies path ending in an atom.

144 3.1.1 The *step* procedure

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

147 Free variables are those variables that appear in a tree; they are used and updated when
148 a backchaining operation takes place.

149 The *step_tag* (see Type 6) indicates the kind of an internal tree step: **CutBrothers** denotes
150 the interpretation of a superficial cut, i.e., a cut whose parent nodes are all *And*-nodes.
151 **Expanded** denotes the interpretation of non-superficial cuts or predicate calls. **Failure** and
152 **Success** are returned for, respectively, *failed* and *success* trees.

153 The step procedure is intended to interpretate atoms, that is, it transforms the tree iff its
154 *path_end* is an atom, otherwise, it returns the identity.

155 **Lemma** *succ_step_iff* u p fv s A: *success* A <-> *step* u p fv s A = (fv, Success, A). (1)

156 **Lemma** *fail_step_iff* u p fv s A: *failed* A <-> *step* u p fv s A = (fv, Failed, A). (2)

157 *Call step* The interpretation of a call *c* starts by calling the *bc* function on *c*. The output
158 list *l* is taken to represent build the new subtree. If *l* is empty then *KO* tree is returned,
159 otherwise the subtree is a right-skewed tree made of *n* inner *Or* nodes, where *n* is the length
160 of *l*. The root has *KO* as left child. The lhs of the other nodes is a right-skewed tree of *And*
161 nodes. The *And* nodes are again a right-skewed tree containing premises of the selected rule.

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

dire dei reset
point

dire che le
sostituzioni del
backchain sono
importanti e

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

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

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

176 ► **Definition 3** (Hard-kill, $!_r$). *Given a tree t , hard-kill replaces the given subtree with the*
 177 *KO node*

178 ► **Definition 4** (Soft-kill, $!_l$). *Given a successful tree t , soft-kill replaces with KO all subtrees*
 179 *that are not part of the path in t leading to the OK node.*

180 An example of the impact of the cut is show in Figure 4c, the dashed triangles represent
 181 generic trees. The step routine interprets the cut since it is the node in its path-end: we pass
 182 through a and and all trees on the left of the cut are successful. In the example we have 4
 183 arrow tagged with the $!_l$ or $!_r$ symbols. The $!_l$ arrows go left and soft-kill the pointed subtree,
 184 it keeps *OK* nodes since they are part of the tree leading to the cut, and replaces the other
 185 subtrees with *KO*. The $!_r$ procedure replaces the nodes pointed by the arrows with *KO*.

186 3.1.2 The *next_alt* procedure

187 It is evident that the *step* alone is not sufficient to reproduce entirely the behavior of the full
 188 expected prolog interpreter. In particular, we need to bracktrack on failures. Moreover, in
 189 case of success, we should return a state where the state is cleaned of the success itself, this
 190 is essential to, non deterministically, find all the solution of a given query. By Lemmas 1
 191 and 2, we know that *step* returns the identity on successful and failed states. In order to
 192 continue the computation on these particular trees, we need the *next_alt* procedure aiming
 193 to expecially work with failed and successful trees: and its implementation in Figure 5.

194 The *next_alt* procedure takes a boolean and a tree, clean it from failures or success and
 195 returns a new tree if this tree still contains a non explored path. The idea behind *next_alt* is
 196 to clean recursively every subtree in DFS order if its *path_end* is a failure. Moreover, if the
 197 boolean passed to *next_alt* is true, then it erases the first successful path in the tree.

198 The base cases of *next_alt* are immediate. The *Or* case is rather intuitive: if the lhs
 199 of the *Or* does not exist we look for the *next_alt* in the rhs. Otherwise, we look for the
 200 *next_alt* in the lhs, if this *next_alt* does not exists, we look for the *next_alt* in the rhs.

201 We want to spend few words about the *And* case, since the reset point $B0$ for B plays an
 202 important role. The *next_alt* in an *And* tree should consider two cases: if the lhs succeeds,
 203 then the *next_alt* should be retrived in the rhs. If this alternative does not exists it means
 204 that the rhs has entirely been explored. We need to erase the success in the lhs and try to
 205 find if a non-explored alternative exists. If so, we return a new tree with the new lhs and the
 206 rhs is built from the reset point. **big_and** is a trivial function build a right-skewed tree of
 207 and nodes where the leaves are the atoms written in the reset point. We need to reuse the
 208 reset point since, the step procedure in *And* trees evaluates the rhs of a *And* tree if the lhs
 209 succeeds. This evaluation is dependent on the substitution in the lhs tree. Therefore, if we
 210 need to backtrack in the lhs, we need to reset the rhs.

```

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 := And A B0 (big_and B0) in
      if success A then
        match next_alt b B with
        | None => omap build_B0 (next_alt true A)
        | Some B' => Some (And A B0 B')
      end
      else if failed A then omap build_B0 (next_alt false A)
      else Some (And A B0 B)
    | Or None sB B => omap (fun x => Or None sB x) (next_alt b B)
    | Or (Some A) sB B =>
      match next_alt b A with
      | None => omap (fun x => Or None sB x) (next_alt false B)
      | Some A' => Some (Or (Some A') sB B)
    end
  end.

```

■ **Figure 5** *next_alt* implementation

Some interesting property of *next_alt* are shown below and allow to see how *next_alt* complements *step*.

Lemma *path_atom_next_alt_id* $b A$: $\text{path_atom } A \rightarrow \text{next_alt } b A = \text{Some } A$. (3)

Lemma *next_alt_failedF* $b A A'$: $\text{next_alt } b A = \text{Some } A' \rightarrow \text{failed } A' = \text{false}$. (4)

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 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 from
the or

3.1.3 The *run* inductive

The inductive procedure *run* is modeled as a function: it takes as input a program, a set of free variables, an initial substitution σ_0 , and a tree t_0 , and returns a substitution σ_1 together with an optional updated tree t_1 . The substitution σ_1 represents the most-general unificator that makes the execution of the tree t_0 succeed starting from the initial substitution σ_0 , σ_1 is an extension of σ_0 . The tree t_1 is the updated tree containing the alternatives that have not yet been explored. If the tree contains no solution, then *None* is returned.

The procedure *run* is based on three main derivation rules, shown in Figure 6. If the *path_end* of the tree t is a success, the input substitution is returned and the input tree is cleaned of its successful path. If the *path_end* of the tree is an atom, then *step* is invoked to evaluate this atom, and *run* is recursively called on the new tree. Finally, if the *path_end* of the tree is a failure, *next_alt* is called to clear the failed path; if the resulting cleaned tree exists, *run* is recursively called on it.

$$\begin{array}{c}
\frac{\text{success } A}{\text{run } fv \ s_1 \ A \ (\text{get_subst } s_1 \ A) \ (\text{next_alt } \top \ A) \ \text{run_done}} \\
\\
\frac{\text{path_atom } A \quad \text{step } u \ p \ fv_0 \ s_1 \ A = (fv_1, \text{st}, B) \quad \text{run } fv_1 \ s_1 \ B \ s_2 \ r}{\text{run } fv_0 \ s_1 \ A \ s_2 \ r} \text{run_step} \\
\\
\frac{\text{failed } A \quad \text{next_alt } \perp \ A = \text{Some } B \quad \text{run } fv_0 \ s_1 \ B \ s_2 \ r}{\text{run } fv_0 \ s_1 \ A \ s_2 \ r} \text{run_fail}
\end{array}$$

■ **Figure 6** Rule system for *run*

3.1.4 Valid tree

The inductive tree allows one to generate a large number of trees, some of which are not valid, in the sense that they cannot be produced starting from a given query. The class of valid trees is characterized by the following function.

```

Fixpoint valid_tree s :=
  match s with
  | TA _ | OK | KO => true
  | Or None _ B => valid_tree B
  | Or (Some A) _ B => valid_tree A && ((B == KO) || B.base_or B)
  | And A B0 B => valid_tree A &&
    if success A then valid_tree B
    else B == big_and B0
  end.

```

Once again, the most interesting cases to analyze are *Or* and *And*.

For the *Or* constructor, we distinguish two cases depending on whether the left-hand side (lhs) exists. If it does not exist, then the right-hand side (rhs) must be a valid tree. Otherwise, the lhs must itself be a valid tree, and the rhs is either the *KO* tree, since it may have been removed by the evaluation of a superficial cut in the lhs, or it has not yet been explored. In the latter case, it is a *base_or* tree, namely the right-skewed tree formed by a disjunction of conjunctions.

For the *And* constructor, the lhs is required to be a valid tree. The shape of the rhs depends on whether the lhs represents a success. If the lhs is not successful, then the rhs has never been explored: the procedures *step* and *next_alt* modify the rhs only when the lhs succeeds. In this case, the lhs must be the right-skewed tree containing the conjunctions of the atoms present in the reset point B_0 . In other words, the rhs coincides with the reset point. If the lhs is a success tree, then the rhs must be a valid tree.

3.2 Elpi semantics

We now want to introduce the elpi semantics. The interpreter we show reflects the interpreter of the ELPI language and is 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]. Pusch 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 semantics is defined as follows:

```

Inductive alts := no_alt | more_alt of (Sigma * goals) & alts
with goals := no_goals | more_goals of (A * alts) & goals .

```

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

263 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.

```

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

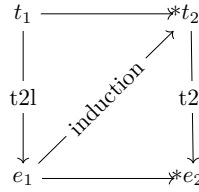
```

Fixpoint t2l (A: tree) s (bt : alts) : alts :=
  match A with
  | OK          => [:: (s, [::])] ]
  | KO          => [::]
  | TA a        => [:: (s, [:: (a, [::])] )] ]
  | Or None s1 B => add_ca_deep bt (t2l B s1 [::])
  | Or (Some 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 := t2l B0 in
    let lA := t2l 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 := map (catr lB0) xs in
      let lB := t2l B s1A (xs ++ bt) in
      (map (catl xz) lB) ++ xs
    else [::]
  end.

```

► **Theorem 5** (tree_to_elpi).

265 $\forall A \sigma_1 B \sigma_2 b \sigma_0, \forall t A \rightarrow$
 266 $\text{run}_u \sigma_1 A (\text{Some } \sigma_2) B b \rightarrow$
 267 $\exists x xs, t2l A \sigma_1 \emptyset = x :: xs \wedge \text{nur}_u x.1 x.2 xs \sigma_2 (t2l B \sigma_0 \emptyset).$



■ **Figure 7** Induction scheme for Theorem 6

► **Theorem 6** (`elpi_to_tree`).

$$\begin{aligned}
& \forall \sigma_1 \sigma_2 a na g, \\
& \text{nur}_u \sigma_1 g a \sigma_2 na \rightarrow \\
& \forall \sigma_0 t, \text{vt } t \rightarrow (t2l \ t \ \sigma_0 \ \emptyset) = ((\sigma_1, g) :: a) \rightarrow \\
& \exists t' n, \text{run}_u \sigma_0 t \ (\text{Some } \sigma_2) \ t' n \wedge t2l \ t' \ \sigma_0 \ \emptyset = na.
\end{aligned}$$

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 `12t` 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 `12t(e)`. However, it is difficult to retrieve 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.

- 304 **7** Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast,
305 embeddable, λ Prolog interpreter. In *Proceedings of LPAR*, volume 9450 of *LNCS*, pages
306 460–468. Springer, 2015. URL: <https://inria.hal.science/hal-01176856v1>, doi:10.1007/
307 978-3-662-48899-7_32.
- 308 **8** Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq*
309 *Workshop*, July 2023. URL: <https://inria.hal.science/hal-04467855>.
- 310 **9** Davide Fissore and Enrico Tassi. Higher-order unification for free!: Reusing the meta-
311 language unification for the object language. In *Proceedings of PPDP*, pages 1–13. ACM, 2024.
312 doi:10.1145/3678232.3678233.
- 313 **10** Davide Fissore and Enrico Tassi. Determinacy checking for elpi: an higher-order logic program-
314 ming language with cut. In *Practical Aspects of Declarative Languages: 28th International*
315 *Symposium, PADL 2026, Rennes, France, January 12–13, 2026, Proceedings*, pages 77–95,
316 Berlin, Heidelberg, 2026. Springer-Verlag. doi:10.1007/978-3-032-15981-6_5.
- 317 **11** Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality
318 tests, automatically. In *Proceedings of CPP*, page 167–181. Association for Computing
319 Machinery, 2023. doi:10.1145/3573105.3575683.
- 320 **12** Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing type theory
321 in higher order constraint logic programming. In *Mathematical Structures in Computer*
322 *Science*, volume 29, pages 1125–1150. Cambridge University Press, 2019. doi:10.1017/
323 S0960129518000427.
- 324 **13** Robbert Krebbers, Luko van der Maas, and Enrico Tassi. Inductive Predicates via Least
325 Fixpoints in Higher-Order Separation Logic. In Yannick Forster and Chantal Keller, editors,
326 *16th International Conference on Interactive Theorem Proving (ITP 2025)*, volume 352 of *Leib-*
327 *niz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:21, Dagstuhl, Germany,
328 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: [https://drops.dagstuhl.](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.27)
329 [de/entities/document/10.4230/LIPIcs.ITP.2025.27](https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2025.27), doi:10.4230/LIPIcs.ITP.2025.27.
- 330 **14** Dale Miller. A logic programming language with lambda-abstraction, function variables, and
331 simple unification. In *Extensions of Logic Programming*, pages 253–281. Springer, 1991.
- 332 **15** Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge
333 University Press, 2012.
- 334 **16** Cornelia Pusch. Verification of compiler correctness for the wam. In Gerhard Goos, Juris
335 Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors,
336 *Theorem Proving in Higher Order Logics*, pages 347–361, Berlin, Heidelberg, 1996. Springer
337 Berlin Heidelberg.
- 338 **17** Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog
339 dialect). In *The Fourth International Workshop on Coq for Programming Languages*, January
340 2018. URL: <https://inria.hal.science/hal-01637063>.
- 341 **18** Enrico Tassi. Deriving proved equality tests in Coq-Elpi. In *Proceedings of ITP*, volume 141 of
342 *LIPIcs*, pages 29:1–29:18, September 2019. URL: <https://inria.hal.science/hal-01897468>,
343 doi:10.4230/LIPIcs.CVIT.2016.23.
- 344 **19** Luko van der Maas. Extending the Iris Proof Mode with inductive predicates using Elpi.
345 Master’s thesis, Radboud University Nijmegen, 2024. doi:10.5281/zenodo.12568604.
- 346 **20** David H.D. Warren. An Abstract Prolog Instruction Set. Technical Report Technical Note 309,
347 SRI International, Artificial Intelligence Center, Computer Science and Technology Division,
348 Menlo Park, CA, USA, October 1983. URL: [https://www.sri.com/wp-content/uploads/](https://www.sri.com/wp-content/uploads/2021/12/641.pdf)
349 [2021/12/641.pdf](https://www.sri.com/wp-content/uploads/2021/12/641.pdf).