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

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:13
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 :=                              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 ℙ := { rules : seq R; sig : sigT }.                              (3)
Definition Σ := {fmap V → Tm}.                                          (4)
Definition bc : Unif → ℙ → $\mathcal{F}_v$ → Callable →
                       Σ → $\mathcal{F}_v$ * seq (Σ * seq A) :=                         (5)
```

!!!: controllare il tipo di bc nel testo

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 → Σ → option Σ;
  matching : Tm → Tm → Σ → option Σ;
}.
```

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

se metti r1 = g A B :- f A B. allora g e f sono funzioni, e puoi spiegare anche l'idea del detcheck qui

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.

4 Tree semantics

```
Inductive tree :=
  | KO | OK | TA of A
  | Or  of option tree & Σ & 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. TA = Todo/-Goal?

```
Fixpoint get_end s A : Σ * 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.
```

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

Definition path_end A := (get_end ϵ A).2. (* ~ϵ~ is the ~ϵ~ subs

Definition success A := path_end A == OK.

Definition failed A := path_end A == KO.

Definition path_atom A := if path_end A is TA _ then true else
```

Figure 3 Defintion of *get_end*

non-terminal è roba di grammatiche, usa nodes/constructors

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 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 $\mathit{t2l}(A) = A_1, \ldots, A_n$.

t2l nope, metti un r3 = g X Z :- r .., f .., !, e rifatti all'esempio della sezione prima (fai in modo che f funzioni solo con la seconda regola per r)

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 *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.

associate to the...

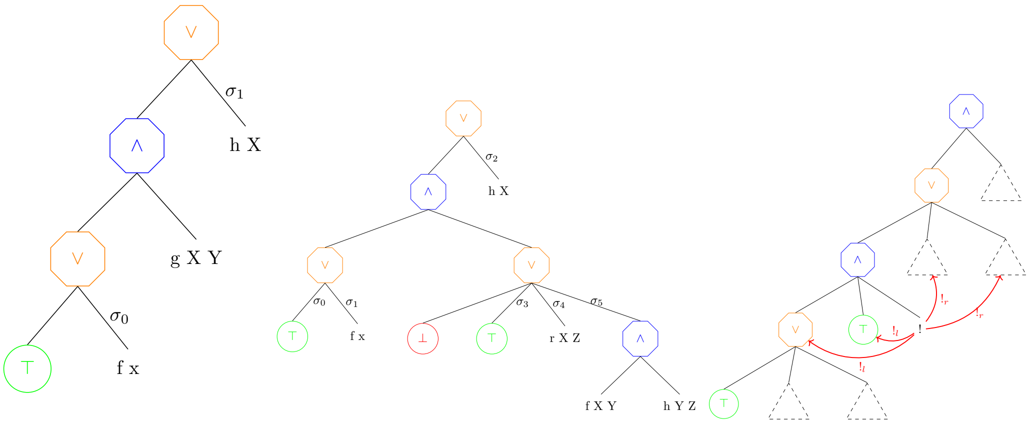
```
Inductive tag := Expanded | CutBrothers | Failed | Success.        (6)
Definition step : ℙ → 𝓕𝓋 → Σ → tree → (𝓕𝓋 * tag * tree) :=        (7)
Definition next_alt : 𝔹 → tree → option tree :=                    (8)
Inductive run (u:Unif) (p : ℙ): 𝓕𝓋 → Σ → tree →
                  Σ → option tree → Prop :=                        (9)
```

as much as needed, indeed prolog programs do not necessarily terminate

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 3, 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 substition 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 ϵ A).2. (* ~ϵ~ is the ~ϵ~ subst *)   (2)
```



(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 Figure 4a the *path_end* of the tree is `g X Y`.

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

```
Definition success A := path_end A == OK.                                   (3)
Definition failed A := path_end A == KO.                                    (4)
Definition path_atom A := if path_end A is TA _ then true else false.       (5)
```

The latter definition identifies path ending in an atom.

4.1 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 and updated when a backchaining operation takes place.

The *step_tag* (see Type 6) indicates the kind of an internal tree step: `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 transforms the tree iff its *path_end* is an atom, otherwsise, it returnes the identity.

```
Lemma succ_step_iff u p fv s A: success A ↔ step u p fv s A = (fv, Success, A).(1)
Lemma fail_step_iff u p fv s A: failed A ↔ step u p fv s A = (fv, Failed, A).  (2)
```

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 . 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 the fourth contains the premises of respectively r_2 and r_3.

dire che le sostituzioni del backchain sono importanti e

Cut step The cut case is delicate, since interpreting a cut in a tree has three main impacts: at first the cut 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** (Hard-kill, $!_r$). *Given a tree t, hard-kill replaces the given subtree with the* KO *node*

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

An example of the impact of the cut is show in Figure 4c, the dashed triangles represent generic trees. The step routine interprets the cut since it is the node in its path-end: we pass throgh a and and all trees on the left of the cut are successful. 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, it keeps *OK* nodes since they are part of the tree leading to the cut, and replaces the other subtrees with *KO*. The $!_r$ procedure replaces the nodes pointed by the arrows with *KO*.

4.2 The *next_alt* procedure

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

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

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

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

```
Definition next_alt : 𝔹 → 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: path_atom A → next_alt b A = Some A.` (3)

`Lemma next_alt_failedF b A A': next_alt b A = Some A' → failed A' = 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 form the or

4.3 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.

$$\frac{\text{success A} \qquad \text{get\_subst } s_1 \text{ A} = s_2 \qquad (\text{next\_alt true A}) = \text{B}}{\text{run fv } s_1 \text{ A } s_2 \text{ B}} \text{RUN\_DONE}$$

$$\frac{\text{path\_atom A} \qquad \text{step u p fv0 } s_1 \text{ A} = (\text{fv1, st, B}) \qquad \text{run fv1 } s_1 \text{ B } s_2 \text{ r}}{\text{run fv0 } s_1 \text{ A } s_2 \text{ r}} \text{RUN\_STEP}$$

$$\frac{\text{failed A} \qquad \text{next\_alt false A} = \text{Some B} \qquad \text{run fv0 } s_1 \text{ B } s_2 \text{ r}}{\text{run fv0 } s_1 \text{ A } s_2 \text{ r}} \text{RUN\_FAIL}$$

Figure 6 Rule system for *run*

5 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].

The inductive representing a state of the Elpi language is shown below.

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

An elpi state is an enhanced two-dimension list. The outermost list represents the list of alternatives in disjunction accompagnate with the substitution that should be used to for their interpretation. The innermost list is a list of atom, representing a list of goals in cunjunctions. These goals are decorated with a pointer to an elpi state, and are used to keep trace of the alternatives that should be kept when a cut is interpreted. We call these, special, alternatives the cut-to alternatives.

The idea of the Elpi interpreter is to receive a list of alternatives. The first alternative consists of a list of goals. Four cases must be taken into account; they are shown in Figure 7. In order to simplify goal retrieval, we split the head of the alternatives from the tail, so that it can be immediately matched in the inductive definition. Note that an empty list of alternatives represents, by definition, a failing state. If the goal list is empty (StopE), then we have, by definition, a success, and the input solution together with the list of alternatives is returned. If the goal list starts with a cut (CutE), then the current alternatives are erased in favour of the cut alternatives, and a recursive call is made on the remaining goal list.

Finally, we must consider the case in which the goal list starts with a call. The call can either fail (FailE) or succeed (CallE). We distinguish the two cases by looking if the backchaining operation returns zero or more rules. We have wrapped this task in the *stepE* procedure, which also updates the goal and cut-alternative list. The fail case, is relatively easy: the first goal does not succeed, we need to take the head of the alternatives, and make it the new list of goals to be explored.

The case in which backchaining produces a non empty list, the *save_alts* routine is in cahrge of: taking the list of premises and add to each atom the the list of alternatives a as their new cut-alternatives, then it append the list of goals gl to each of these new lists.

```
Definition stepE fv t s a gl :=
  let (fv', rs) := bc u p fv t s  in
  let rs_ca := save_alts a gl (r2a rs) in
  (fv', rs_ca).

Inductive nur : F_V → Σ → goals →  alts → Σ → alts → Prop :=        (10)
```

$$\frac{}{\text{nur fv s [::] a s a}}\ \textsc{StopE}$$

$$\frac{\text{nur fv s gl ca } s_1 \text{ r}}{\text{nur fv s [:: (cut, ca) \& gl] a } s_1 \text{ r}}\ \textsc{CutE}$$

$$\frac{\text{stepE fv t s al gl} = \text{(fv', [:: b \& bs])} \qquad \text{nur fv' b.1 b.2 (bs++al) } s_1 \text{ r}}{\text{nur fv s [:: (call t, ca) \& gl] al } s_1 \text{ r}}\ \textsc{CallE}$$

$$\frac{\text{stepE fv t s al gl} = \text{(fv', [::])} \qquad \text{nur fv' } s_1 \text{ a al } s_2 \text{ r}}{\text{nur fv s [:: (call t, ca) \& gl] [:: } (s_1, \text{a}) \text{ \& al] } s_2 \text{ r}}\ \textsc{FailE}$$

Figure 7 Rule system for *nur*

6 Semantic equivalence

The equivalence between the two semantics is possible under two conditions: we need to work with "valid states", i.e. all of those states that can be generated from a call. Secondly, we need to translate trees state into elpi states. In the next to subsection we propose the two functions *valid_state* and *t2l*.

6.1 Valid trees

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 function shown in Figure 8a.

While all base cases of tree are considered valid, we need to analyze carefully the cases for the *Or* and *And* constructors.

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 that is genereated after a succeful backchain to a call.

For the *And* constructor, the lhs is required to be a valid tree. The shape of the rhs depends on whether the lhs is 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 conjunctions of premises atoms, the reset point B_0 ensure what shape the rhs should have. If the lhs is a success tree, then the rhs can be modified by *step* and *next_alt*, therefore it must be a valid tree.

```
Fixpoint valid_tree A :=
  match A 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.
```

(a) Valid tree

```
Fixpoint t2l A s0 bt :=
  match A with
  | OK           => [:: (s0, [::])]
  | KO           => [::]
  | TA a         => [:: (s0, [:: (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 s0 lB in
      add_ca_deep bt (lA ++ lB)
  | And A B0 B   =>
      let lA  := t2l A s0 bt in
      let lB0 := a2g B0 in
      let lA  := add_deep bt lB0 lA in
      if lA is [:: (s0, gs) & al] then
        let al := map (catr lB0) al in
        let lB := t2l B s0 (al ++ bt) in
        map (catl gs) lB ++ al
      else [::]
end.
```

(b) Tree to list

Figure 8 Valid tree and Tree to list

6.2 From trees to lists

The translation of a tree to a list is shown in Figure 8b. It takes the tree to be translated, a substitution (called s), a list of alternatives, called *bt*. The substitution s tells what is the subsitution of the alternative we are building and is updated when going to the rhs of the *Or* constructor. The *bt* list represents the future alternatives list that have the same depth of the current tree root. They are useful to know if they should be added to the current goal as cut alternatives.

An *OK* node represent a success in a tree, then the translation of this state returns a singleton list with the cuple $(s, [::])$: there is one alternative with 0 goals, i.e. a success in elpi by StopE. The *KO* node represent a failure, and, therefore 0 alternatives. An atom is translated into a singleton with the atom as first goal and the empty cut-alternative list: we are not adding *bt* since they are the alternatives for the right-sibling, not the right-uncles, this means that if we have a is a cut, then it erases the alternatives bt.

In a disjunction, we are scendendo di un livello the distance from our backtracking list. This means that *bt* becomes the list of right-uncles of the two branches of the *Or* constructor. The compilation of the rhs is done independently of if the lhs exists: we transform the rhs into an elpi state and passing the empty list of alternatives, since the rhs as no right-siblings. Then, thanks to `add_ca_deep`, we recursively add *bt* to every cut-alternative appearing in the translated goals. If the lhs exists, we translate it and pass the translation of the rhs as the list of right-siblings. Then we prepend the translated lhs to the translated rhs and add *bt* with `add_ca_deep`.

We compile the *And* case by translating its lhs to the list lA and reset point to the list $lB0$. Then, thanks to `add_deep`, we recrusively append $lB0$ to the alternatives born in lA, i.e. we leave unchanged the pointers to *bt*, if any, in lA. We finally need to treat cases whenever the list lA is empty or not. In the former case the empty list is returned, otherwise,

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

Figure 9 Induction scheme for Lemma 6

we hasve a list $[:: (s_0, gs)\&al]$. By the semantics of the *And* in the *run* interpreter, the rhs of the *And* represent the sequent of goals to be executed after the first alternative in the lhs, it corresponds to the list of goals gs. The reset point $B0$ is to be appended to the next alternatives in the lhs, that is the queue of lA, that is al.

alternative in tree? rephrase

We note therefore that the compilation of lB is done by passing the alternatives $al++bt$ since the alternatives in al are to be...

6.3 Equivalence theorems

```
Lemma tree_to_elpi: ∀ u p fv s0 t s2 t',
  vars_tree t `<=` fv → vars_sigma s0 `<=` fv →
  valid_tree t →
    run u p fv s0 t s2 t' →
      ∃ na s1 g a,
        t2l (odflt KO t') s0 [::] = na /\
        t2l t s0 [::] = (s1,g) :: a /\
          nur u p fv s1 g a s2 na.
```
(5)

```
Lemma elpi_to_tree: ∀ u p fv s1 g s2 a na,
  nur u p fv s1 g a s2 na →
    ∀ s0 t, valid_tree t → t2l t s0 [::] = (s1,g) :: a →
      ∃ t', run u p fv s0 t s2 t' /\ t2l (odflt KO t') s0 [::] = na.
```
(6)

The proof of Lemma 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 $\texttt{l2t}(e)$. 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:

7 Case study: determinacy alanysis

we mechanize the first order part of xxx.

snippet det, main thm, invariant det tree (valid tree prev section?)

proof induciton on exec, step/next alt preserving invariant proved by induction on the tree.

with list semantics cut and next alt requires to express a link btween the ca or next alts and the current goal, which is non trivial without an intermediate data strature like the tree

8 Related work

prolog semantics, King lost

yves for the proof technique

9 Conclusion

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.