Dummy title

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 Common code: the language

```
Inductive Tm :=
  | Tm_Kp    : Kp -> Tm
  | Tm_Kd    : Kd -> Tm
  | Tm_V     : V  -> Tm
  | Tm_Comb  : Tm -> Tm -> Tm.

Inductive Callable :=
  | Callable_Kp   : Kp -> Callable
  | Callable_V    : V -> Callable
  | Callable_Comb : Callable -> Tm -> Callable.

Inductive RCallable :=
  | RCallable_Kp   : Kp -> RCallable
  | RCallable_Comb : RCallable -> Tm -> RCallable.
```

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

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

An atom is the smallest syntactic unit that can be executed in a prolog program $\mathcal{P}$.

```
Record R := mkR { head : RCallable; premises : list A }.
```

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

[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. 23:1–23:5
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
(*simpler than in the code: signatures of preds are hidden*)
Definition program := seq R.
```

A program is made by a list of rules. Rules in $\mathcal{P}$ are indexed by their position in the list. Given a list of rules $\mathcal{R}$and two indexes i and j, s.t. $i \neq j$ then, $\mathcal{R}_i$ has a higher priority then $\mathcal{R}_j$.

Sigma is a substitution mapping variables to their term instantiation.

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

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

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

2 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 to have a graphical view of its evaloution while the progrma is being intepreted. The second syntax is the elpi's syntax, we call it therefore elpi. We aim to prove the equivalence of the two semantics together with some interesting lemmas of the cut behavior.

2.1 Tree semantics

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

In the tree we distinguish 6 main cases: Bot and OK are respectively the standard fail $\bot$ and true $\top$ predicates of prolog. Dead is a special symbol representing a ghost state, that is, a state useful to keep the structure of a tree from an execution to another but that is completely ignored by the intepretation of the program.

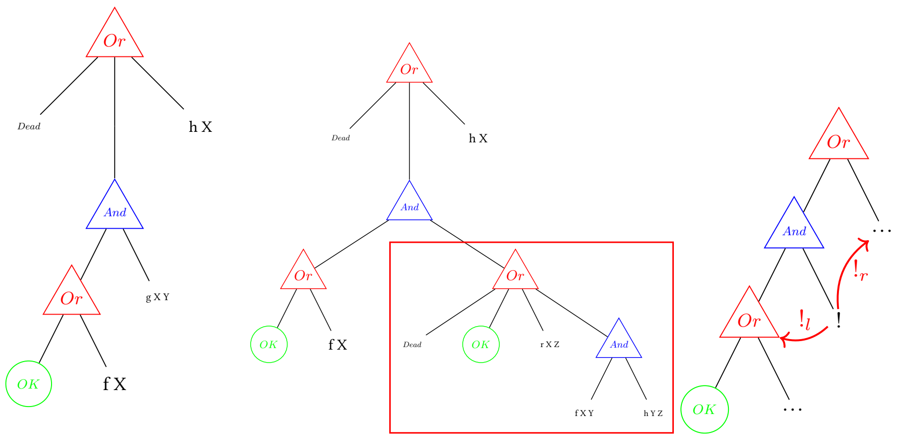
TA, standing for tree-atom, is a terminal of the tree containg an atom.

The two recursive cases of a tree are the Or and the And non-terinals. The Or non-terminals $A \vee B_\sigma$ stands for a disjunction between two trees A and B. The second tree branch is decorated with a suspended substituion σ so that, when we backtrack to B, we use σ as initial substitution for B.

The And non-terminal $A \wedge_r B$ represents of a conjunction of two trees A and B. We call r the reset-point and is used to resume the B state in its intial form if some backtracking operation is performed on A.

The interpretation of a tree is performed by two main routines: `step` and `next_alt` that traverse the tree depth-first, left-to-right.

We get the first to-be-explored terminal in the tree by getting the end of a path. This path is created from a tree traversal starting from the roots and immidiatly ends if the tree is not niether a disjunction, nor a conjunction: the to-be-explored terminal is the tree itself. Otherwise, if the tree is a disjunction, the path continues on the left- or the right-subtree depending of if the path of the lhs is a dead node. In the case of a conjunction, we look for the path of the lhs. If this path returns a success, we build a path in the rhs, otherwise, we return the lhs. In Figure 1a the first non-explored node is g X.



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

Figure 1 Tree with first non explored node g X

The `step` procedure takes a tree and explores it using the path strategy. A success (i.e. a tree with path ending with OK) and failed tree (i.e. a tree with path ending with KO or Dead) is returned as it. The two interesting cases are when the path ends with a call or a cut.

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

```
g X X.                    % r1
g X Z :- r X Z.           % r2
g X Z :- f X Y, h Y Z.    % r3
```

A step in the tree in Figure 1a make a backchain operation over the query g X Y and, in the program above, the new tree would be the one in Figure 1b. We have put a red border aroung the new generated subtree. It is a disjunction of four subtrees: the first node is the Dead node (by default), the second is OK, since r1 has no premises, the third and the fourth contains the premises of respectively r2 and r3.

Cut step The latter case is more delicate since performing a cut in a tree has

dire che le sostituzioni del backchain sono importanti

2.1.1 Valid tree

2.2 Elpi semantics

The Elpi interpreter is based on an operational semantics close to the one picked by Pusch in [4], in turn closely related to the one given by Debray and Mishra in [3, Section 4.3]. Push mechanized the semantics in Isabelle/HOL together with some optimizations that are present in the Warren Abstract Machine [5, 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 G :=
  | callE : Callable -> G
  | cutE : alts -> G
with alts :=
  | no_alt
  | more_alt : (Sigma * goals) -> alts -> alts
with goals :=
  | no_goals
  | more_goals : G -> 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) ++
        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,
```

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

```
Fixpoint t2l (A: tree) s (bt : alts) : alts :=
match A with
| OK => (s, nilC) ::: nilC
| Bot => nilC
| Dead => nilC
| TA cut => (s, ((cutE nilC) ::: nilC)) ::: nilC
| TA (call t) => (s, ((callE t) ::: nilC)) ::: nilC
| Or A s1 B =>
  let lB := t2l B s1 nilC in
  let lA := t2l A s lB in
  add_ca_deep bt (lA ++ lB)
| And A B0 B =>
  let hd  := r2l B0 in
  let lA  := t2l A s bt in
  if lA is more_alt (slA, x) xs then
      let xz := add_deepG bt hd x in
      let xs := add_deep bt hd xs in
      let xs := make_lB0 xs hd in
      let lB   := t2l B slA (xs ++ bt) in
      (make_lB01 lB xz) ++ xs
  else nilC
end.
```

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

Figure 2 Induction scheme for Theorem 2

▶ **Theorem 1** (`tree_to_elpi`).

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

▶ **Theorem 2** (`elpi_to_tree`).

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

The proof of Theorem 2 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 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`.

4 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.

5 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`.