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 $\mathcal{P}$. The execution of an atom, inside a program and a substitution either succeeds returining an output substitution, or it fails. In both cases it returns a list of choice points, representing suspending states that can be resumed for backtracking.

[1] Optional footnote, e.g. to mark corresponding author

```
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.

```
(*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 and a program.

The two recursive cases of a tree are the Or and the And non-terinals. The Or non-terminals $A \lor B_\sigma$ 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 $A \land_r B$ 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 main

```
Inductive step_tag := Expanded | CutBrothers| Failure | Success.


Fixpoint step pr s A : (step_tag * tree) :=
  let step := step pr in
  match A with
  | OK             => (Success, OK)
  | Bot | Dead     => (Failure, A)

  | TA cut       => (CutBrothers, OK)
  | TA (call t) => (Expanded, (big_or pr s t))

  | Or A sB B =>
      if is_dead A then
        let rB := (step sB B) in
        (if is_cb rB.1 then Expanded else rB.1, Or A sB rB.2)
      else
      let rA := step s A in
      (if is_cb rA.1 then Expanded else rA.1, Or rA.2 sB (if is_cb rA.1 then cutr B else B))
  | And A B0 B =>
      let rA := step s A in
      if is_sc rA.1 then
        let rB := (step (get_substS s rA.2) B) in
        (rB.1, And (if is_cb rB.1 then cutl A else A) B0 rB.2)
      else (rA.1, And rA.2 B0 B)
  end.
```

Figure 1 Step for tree semantics

```
Fixpoint next_alt b (A : tree) : option (tree) :=
  match A with
  | Bot | Dead => 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 is_dead A then omap (fun x => (Or A sB x)) (next_alt b B)
    else match next_alt b A with
      | None => obind (fun x => Some (Or (dead A) sB x)) (next_alt false B)
      | Some nA => Some (Or nA sB B)
    end
end.
```

Figure 2 backtracking operation

TODO: define path

The tree interpreter is made by two fixpoints and an inductive.

We make the distinction between some kind of particular trees:

1. success is a tree with a successfull path
2. failed is a tree with a failed path
3. dead is a tree with deads states

2.2 Elpi semantics

The Elpi interpreter is based on an operational semantics close to the one picked by Pusch in [3], in turn closely related to the one given by Debray and Mishra in [2, Section 4.3]. Push mechanized the semantics in Isabelle/HOL together with some optimizations that are present in the Warren Abstract Machine [4, 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 .
```

```
Fixpoint valid_tree s :=
  match s with
  | TA _ | OK | Bot => true
  | Dead => false
  | Or A _ B =>
    if is_dead A then valid_tree B
    else valid_tree A && (B.bbOr B)
  | And A B0 B =>
    valid_tree A &&
      if success A then valid_tree B
      else B == big_and B0
  end.
```

$$\texttt{B.bbOr}\ A \iff \exists\ r\ rs, A = \texttt{big\_or\_aux}\ r\ rs \lor A = \texttt{cutr}(\texttt{big\_or\_aux}\ r\ rs)$$

Figure 3 Valid tree

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, 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.
```

▶ **Theorem 1** (tree_to_elpi).

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

▶ **Theorem 2** (elpi_to_tree).

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

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 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.

3 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.

4 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.