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 ...

```
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.

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

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

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.
```

Sigma is a substitution mapping variables to their term instantiation.

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

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.

```
Inductive tree :=
  | Bot | OK | Dead
  | TA : program  -> A -> tree
  | Or  : tree -> (Sigma * tree) -> tree
  | And : tree -> (program * 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 $A \vee B_\sigma$ 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.

We make the distinction between some kind of particular trees:

1. success is a tree with containing a successfull path

```
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 && (bbOr B)
  | And A B0 B =>
    valid_tree A &&
      if success A then valid_tree B
      else
        let B' := big_and B0.1 B0.2 in
        B == B'
  end.
```