HAL open science

Optimization Strategy apply to an Attribute Algorithm for the Abstract Interpretation

Kaninda Musumbu

To cite this version:

Kaninda Musumbu. Optimization Strategy apply to an Attribute Algorithm for the Abstract Interpretation. The 5the Indian Conference on Artificial Intelligence, Dec 2011, Siddaganga Institute of Technology, Tumkur, India. pp.618-631. hal-00712076

HAL Id: hal-00712076
https://hal.science/hal-00712076

Submitted on 26 Jun 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimization Strategy apply to an Attribute Algorithm for the Abstract Interpretation

Kaninda MUSUMBU
LaBRI (UMR 5800 du CNRS)
Université Bordeaux I
351, cours de la Libération,
F-33405 Talence Cedex FRANCE
e-mail: musumbu@labri.fr

Abstract

Presented in this work is a novel approach to implement an efficient abstract interpretation algorithm for logic programs by means of attributed grammar. Successful implementation of this algorithm yielded memory storage saving and run-time reduced. We first execute the algorithm presented in [2](slightly modified), which generates a tree with four types of nodes. We then store in computer memory only the pertinent subtrees. In other words, after each step, we prune some branches of tree and develop only the subtree which can improve the "oracle" (i.e. set of abstract tuple).

Keywords: Logic programming, static analysis, abstract interpretation, attribute grammar,incremental evaluation.

1 Introduction

The technique of abstract interpretation [5, 3, 10, 6] is a successful framework for constructing analysis of the run-time behavior of programs. For this, at each program point, we compute an abstract substitution that is a correct approximation of the concrete substitutions that occur at run-time.

Attribute grammars are a very interesting tool for computing abstract interpretation. In fact, the syntactical development of programs during their execution can be expressed by a derivation tree *w.r.t.* an universal grammar where the nodes of the tree are labeled by symbols representing call procedures, clauses, and goals. This grammar describes the replacement process of a call procedure by clauses and a clause by a sequence of goals. An abstract substitution is an attribute attached to each node of the derivation tree.

We introduce an efficient and accurate algorithm which, for a given logic program, computes an increasing sequence of finite attributed trees. At each step, the attributed tree is defined by a transformation of the attributed tree of the preceding step. This sequence converges to a fixed point of the transformation and that fixed point is the abstract semantics of the given logic program.

Presented in this work is a novel approach to implement an efficient abstract interpretation algorithm for logic programs by means of attributed grammar. Successful implementation of this algorithm yielded memory storage saving and run-time reduced. We first execute the algorithm presented in [2](slightly modified), which generates a tree with four types of nodes. We then store in computer memory only the pertinent subtrees. In other words, after each step, we prune some branches of tree and develop only the subtree which

can improve the set of abstract tuple Eta.

We hope this approach to will be permit general purpose hardware and a straight forward algorithm to be used for all analysis that previously required special purpose.
The paper is organized as follows. In section 2, we give the context free grammar of Prolog and the abstract domain. Section 3 describes the computing method. While the abstract interpretation algorithm is given in section 4. We give the details about our optimization strategy and optimized algorithm , based on the concept of *useless* sub-tree, in section 5. In section 6, we conclude.

2 Basic Abstract Semantics

2.1 Abstract Syntax

The abstract syntax of the language can be defined by the following grammar:

$$\begin{array}{lll} P & \in & Programs \\ Pr & \in & Procedures \\ C & \in & Clauses \\ SB & \in & Goals \\ B & \in & Atoms\ (or\ Subgoals) \\ f & \in & Functors \\ p & \in & ProcedureNames \\ x_i & \in & ProgramVariables \end{array}$$

$$\begin{array}{lll} P & ::= & <> \mid Pr.P \\ Pr & ::= & <> \mid Pr.C \\ C & ::= & p(x_1\ ,\ \ldots\ ,\ x_n) \leftarrow SB \\ SB & ::= & <> \mid SB.B \\ B & ::= & x_i = x_j \mid x_{i_1} = f(x_{i_2}\ ,\ \ldots\ ,\ x_{i_n}) \mid p(x_{i_1}\ ,\ \ldots\ ,\ x_{i_n}) \end{array}$$

2.2 Attribute Grammar

Both the mechanism of replacements (of procedures by clauses and clauses by goals) and the process of propagation of abstract interpretation at each replacement can be described by an attribute grammar. The grammar gives the syntactic structure of an abstract tree describing the replacements and equations on attributes define how the abstract substitution is transmitted by the replacements from a node to its children. The attribute grammar is: $G =< G, ATT, Q > where$:

-$G =< T, N, ROOT, P >$ is the grammar,
-ATT= $\{\beta_{out}, eta_{out}, ich_{out}, l_var, \beta_{in}, eta_{in}, ich_{in}, anc, phi_{in}, phi_{out}, type\}$
is the set of attributes.
The meaning of these attributes at each node u of the abstract tree is:
β_{out} and β_{in} represent abstract substitutions;
eta_{out} and eta_{in} are sets of triples $(\beta_{in}, p, \beta_{out})$ satisfying the requirements $\beta_{out} = eta(\beta_{in}, p)$;
ich_{out} and ich_{in} are two boolean that indicate the sets eta_{out} and eta_{in} have been modified;

anc is a set of pairs (p, β_{in})
corresponding to the calls procedure
on the left of the node u;
phi_{out} and phi_{in} are two boolean that indicate,
for each production, if we have meet at
its left a node labeled by the non terminal φ;
$type$ gives the type of productions;
-Q is the set of semantics rules between attributes
associated with the productions,

only given the semantics rules of the set Q of productions which are different from the identity equation i.e different from a(x)=b(y), while the rules which consist of an identity equation are only represented by a simple edge between the two attributes (of the identity equation) on the graphs associated with productions cfr figure 9 The reader can find the definitions of the functions Ext_p, $Restr_C$, $Restr_b$, Ext_b, Abi_1, Abi_2 and map in [1, 7].

2.3 Abstract Domain

For each finite set D of program variables, we assume the existence of a *cpo* AS_D whose elements are called abstract substitutions on domain D and denoted by β. We denote by CS_D the set of program substitutions having D as domain. The meaning of each abstract substitution is given through the concretization function: $Cc : AS_D \to P(CS_D)$. The basic abstract semantics uses seven abstract primitive operations.
Let $D = \{x_1, ..., x_n\}$ be a set of variables and P a program using the elements of D. The abstract semantics of P is defined as abstract tuples $(\beta_{in}, p, \beta_{out})$ where $p \in Pred(P)$, the set of predicate symbols occurring in P, and $\beta_{in}, \beta_{out} \in AS_D$. We denote by eta a total and monotonic function $eta : AS_D \times Pred(P) \longrightarrow AS_D$ satisfying:

1. $\forall(\beta_{in}, p),\ \exists!\ \beta_{out} \in AS_D$ s.t
   $eta(\beta_{in}, p) = \beta_{out}$;
2. $\forall(\beta_1, p)\ and\ (\beta_2, p)$,
   $\beta_1 \le \beta_2 \Rightarrow eta(\beta_1, p) \le eta(\beta_2, p)$.

In the abstract interpretation of the program P, we are interested in studying the properties of the variables occurring in P. We consider a query p and information about the variables of p (like mode, type, sharing, etc) expressed in terms of an abstract substitution and denoted by β. Then, our purpose is to determine the behavior of the program P for the pair (β, p) i.e. to compute the "output" abstract substitution corresponding to β. We proceed in two steps for the computation of the "output" abstract substitution. First, we associate an abstract tree with the tree developed by the Prolog compiler in the model of the Prolog procedural semantics, and secondly, we propagate the abstract substitution through the abstract tree as an attribute. Also, we use a set of others attributes, for a technical purpose.

3 Computing Method

Given a program P, a predicate p and an input abstract substitution β for p, we have to compute the output abstract resulting from the execution of P for (β, p). The result is issued all possible execution of P for p with any concrete input θ described by β i.e. we have to consider trees representing all possibles executions of P. For this purpose we construct a sequence of increasing trees which represent all possible executions of P. To do this, we start in associating a development tree with each predicate of program P w.r.t an universal grammar. The development tree describes the replacements of a predicate by the corresponding clauses and of a clause by the corresponding goals.

In fact, we compute on the program P an abstract substitution for the given predicate and substitution (β,p). To avoid infinity tree we maintain an oracle ETA, in which we store all tuple $(\beta_{in}, p, \beta_{out})$.

As an example,we consider the following example of concatenation of two lists:

$$\begin{array}{lll} p(x_1,x_2,x_3) & \leftarrow & x_1 = [\,], \\ & & x_2 = x_3. \\ p(x_1,x_2,x_3) & \leftarrow & x_3 = [x_4 \mid x_6], \\ & & p(x_5,x_2,x_6), \\ & & x_1 = [x_4 \mid x_5]. \end{array}$$

The function development-tree τ_p associated with the predicate p is showed in figure 1

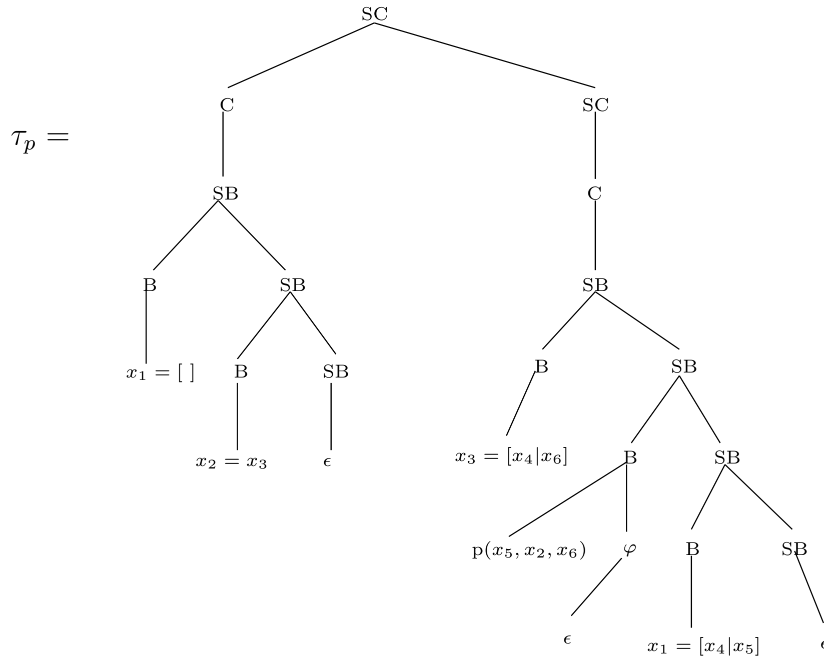

Figure 1: Function development tree

For the sequence of increasing trees, we start with the initial tree in figure 2 with θ_0

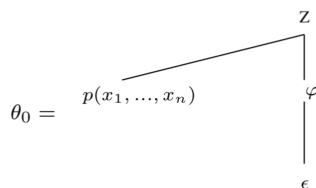

Figure 2: Initial development tree

To define θ_{i+1} from θ_i we replace some sub-trees composed only by the production r_3 (then, the non-terminal at these nodes is ϕ) by the development trees of corresponding predicate (i.e predicate(ϕ)). We make this replacement if the attributes of ϕ satisfy the condition that the pair (predicate(ϕ), $\beta_{in}(\phi)$) don't belong to the set eta_{in}. We give now the procedure which calculates the sequence of increasing trees θ_i :

Procedure developp_tree(θ : in derivation tree, θ' : out derivation tree)
/* This procedure goes all over the nodes needed to be developed */

{ let $u_1, ..., u_n$ be all the nodes of θ_i from left to right

and labeled by the production r_3

$\theta' = \theta$;

for $u = u_1$ to u_n do

{ if (predicate(u), β_{in}(u))$\in$ eta_{in}(u) then

develop_node(θ',u)

}

}

Procedure develop_node(θ : in out derivation tree, u : in node)
/* This procedure substitutes the sub-tree at u in θ by the tree development $\tau_predicate(u)$, where predicate(u) is the value of the attribute predicate at u in *theta* */

{ $\theta = \theta$[u/$\tau_predicate(u)$]
}

where $\theta = \theta$[u/t] is the substitution of the sub-tree at u in θ by the tree t.

As example, we give a sequence without the computation of the attributes see figure 3.

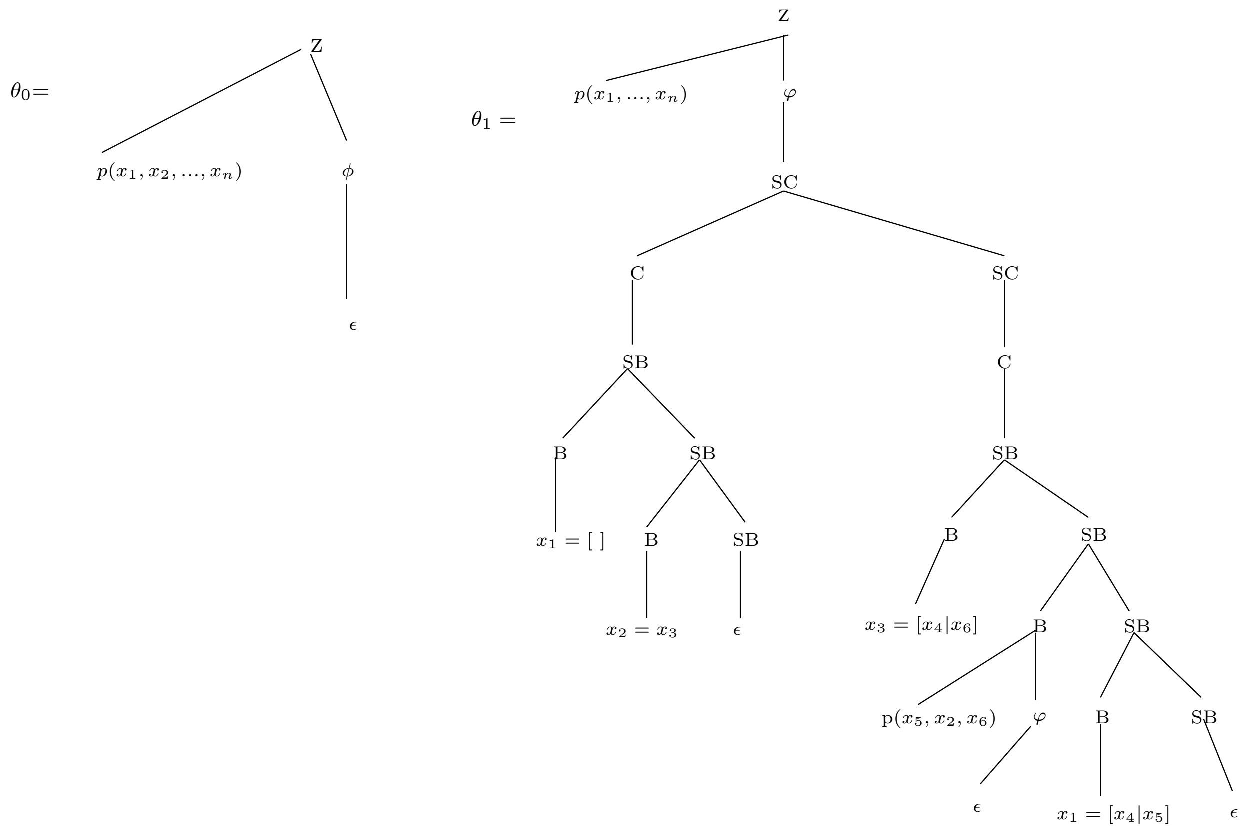

Figure 3: Development tree

4 Abstract Interpretation Algorithm

The evaluation of attributes of each tree of the sequence starts by giving values to the inherited attributes of the root. After, we apply a classical attribute evaluation algorithm in order to compute all attributes of the tree and in particular the synthesized attributes of the root. The value of the input abstract substitution (β_{in}) is the same for all trees. While the input set of eta_{in} of the tree θ_{i+1} takes the value of the output set of eta_{out} of the tree θ_i. Then the sequence of increasing trees converges to a fixed point when the attributes eta_{in} and eta_{out} of the root of some tree θ_i are equal. Finally, the abstract interpretation algorithm alternates the two operations of tree development and attribute evaluation as follows.

Procedure Abstract_Interpretation(p : predicate, β : abstract substitution)
/* This procedure initialize the computation of the abstract substitution */

{ i = 0; /* θ_0 is the initial tree (see ......) */
 Initialize the inherited attributes of the root of θ_0 :
        $\beta_{in}(\text{root}(\theta_0)) = \beta$;
        $eta_{in}(\text{root}(\theta_0)) = \emptyset$;
  Loop_Evaluate_&_Develop();
}

Procedure Loop_Evaluate_&_Develop()
/* This procedure constructs the sequence of increasing trees θ_i and call the procedure *Eval_lr* to compute the attributes of each θ_i*/

{ Eval_lr(θ_i,root(θ_i),$\{\beta_{in}, eta_{in}, ich_{in}, anc, predicate\}$,
        $\{\beta_{out}, eta_{out}, ich_{out}\}$);
  if $eta_{in}(\text{root}(\theta_i)) = eta_{out}(\theta_i))$
        then { /* θ_i is the fixed point */
                print(eta_{out}(root(θ_i));
                exit;
                }
        else { /*the construction of the tree θ_{i+1} */
                develop_tree(θ_i,θ_{i+1});
                $eta_{in}(\text{root}(\theta_{i+1})) = eta_{out}(\text{root}(\theta_i))$
                i = i+1;
                }
}

Procedure Eval_lr(t : in tree, u : in node, I : out set of inherited attributes, S : out set of synthesized attributes,)
/* This procedure computes from left to right all the attributes of I and S of the sub-tree at u in t */

{ let $u_1, ..., u_n$ the sons of u in t from left to right */
  for v = u_1 to u_n do
        {   calculate the inherited attributes I of v;
            Eval_lr(t,v,I,S);
            calculate the synthesized attributes S of v;
        }
  calculate the synthesized attributes S of u;
}

This algorithm terminates since the sequence of values of the attribute eta_{out} is growing and belongs to an inductive domain. But, the main problem of this algorithm is the reevaluation of some attributes common to a sub-sequence of trees. This point will be studied in the following section.

5 Optimization Strategy

The procedure Eval of attribute evaluation, of the preceding section, shows a classical strategy of evaluation for attributes. It consists of visiting all nodes of the attributes tree (θ_i) and evaluating all attributes of each node. We observe that some sub-trees of the attributed tree θ_i are also sub-trees of θ_{i+1} and have the same values for all their attributes. Such sub-trees will be called **useless**.

5.1 Characterization of Useless Sub-trees

A sub-tree of a tree θ_i will be called terminal if it don't contain any occurrence of the production r_3 ($\phi \rightarrow \epsilon$). The useless sub-tree is defined as follows. Let be the following tree θ_i with the sub-tree t_C generated by the non-terminal C :

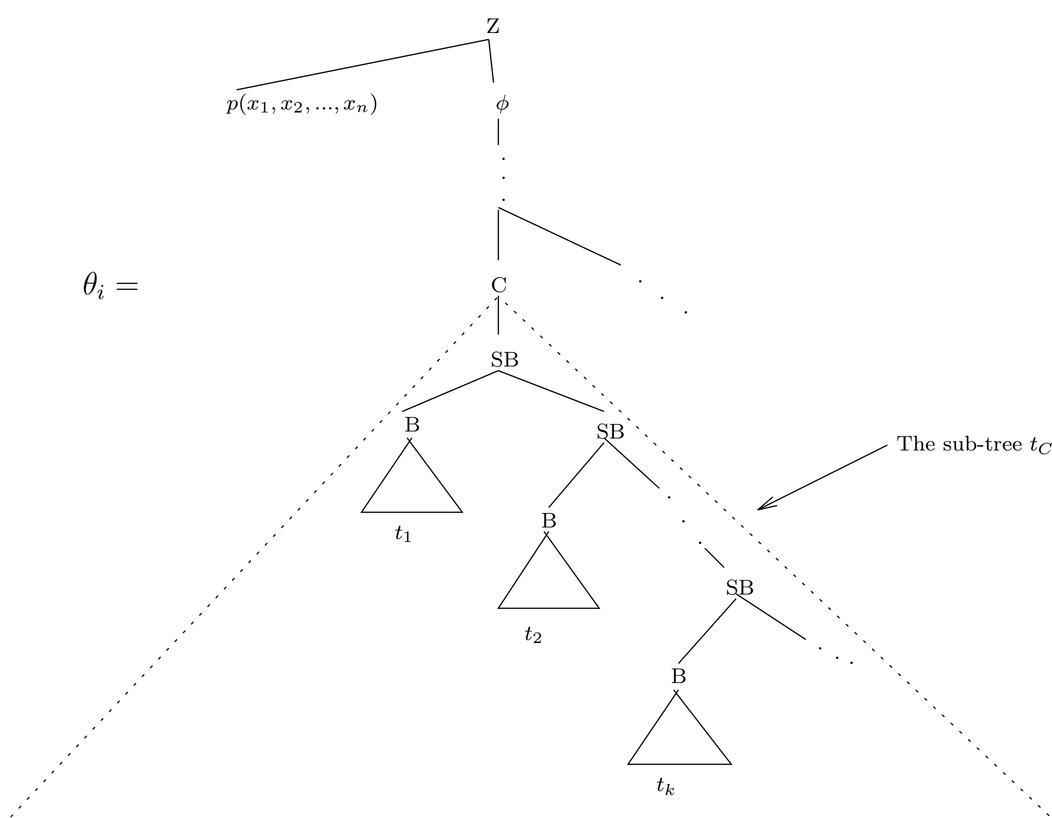

Figure 4: Useless sub-tree

The sub-tree t_k of θ_i is useless (then it don't need to be reevaluated among the tree θ_{i+1}) if all the sub-trees t_1, ... , t_{k-1} on his left and inside the sub-tree t_C are terminal.
Then, to know if t_k is useless or not, we have to propagate an information from t_1, ... , t_{k-1} to the sub-tree t_k which indicates if all these sub-trees are terminal or not. This information is the value of control attribute attached to each node in the tree. We can observe also that the propagation of the values of control attributes through a tree is from left to right. As an example, we take the following tree :

There, the sub-trees t_1 t_2, and t_3 are useless, while the sub-trees t_4 and t_5 must be preserved.

5.2 Propagation of values of control attributes

Two control attributes are needed to transmit the information about the existence of not terminal sub-tree from left to right and into two directions:
- up-down, with an inherited attribute
- bottom-up, with a synthesized attribute.
We want to show by examples the propagation of the values of control attributes. For that, we distinguish two types of sub-trees:
- those rooted by the goal symbol B and called goaled sub-trees;
- those rooted by the clause symbol C and called claused sub-trees.
Now, we present three cases of control attribute propagation which give principal ways of propagation through different sub-trees:

case 1: propagation between goaled sub-trees:

The sub-tree t_k is not terminal and this information (the *true* value) is transmitted to the sub-trees t_{k+1},

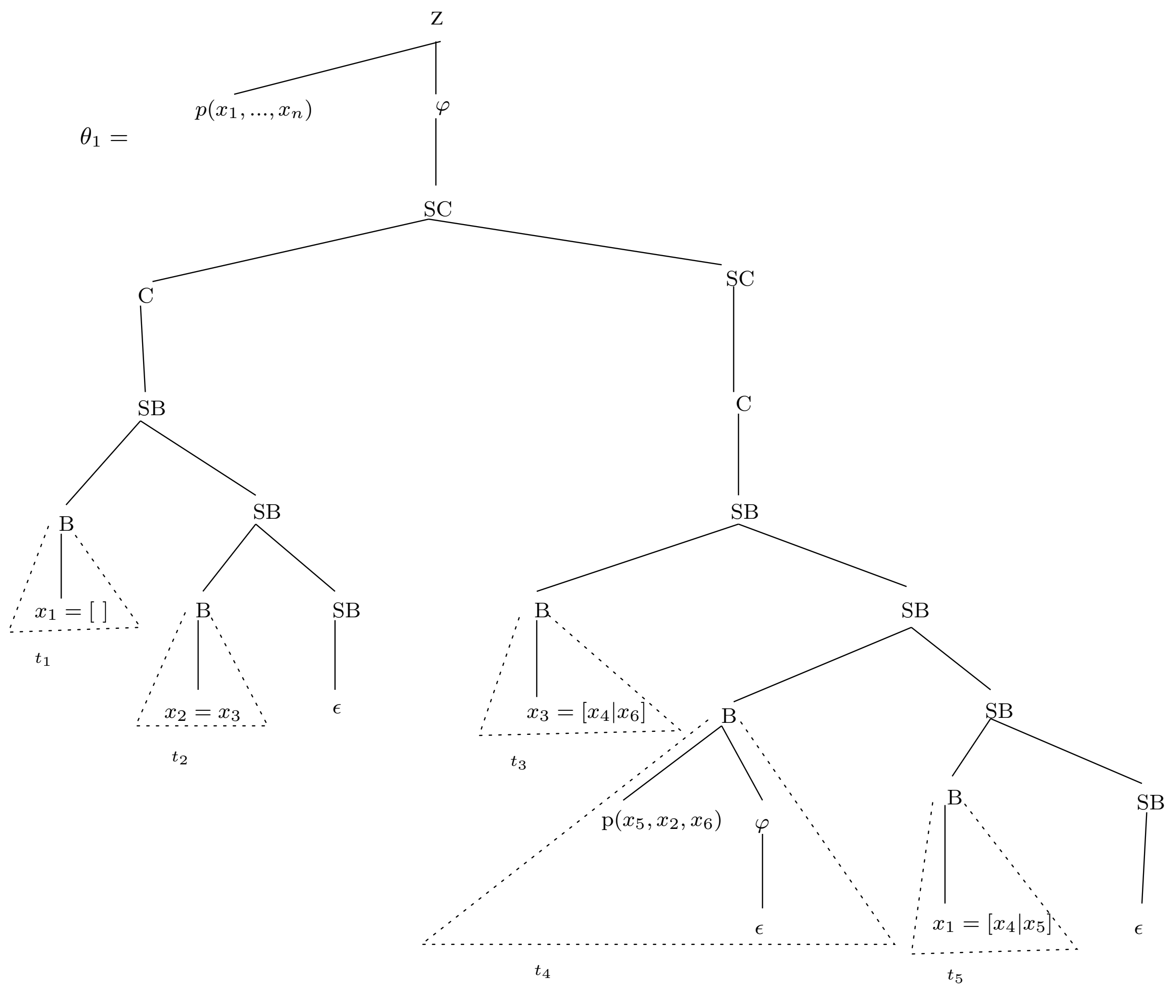

Figure 5: Useless sub-tree

t_{k+2}, ... The sub-tree t_{k+1} depends on t_k and t_{k+2} depends on t_k and t_{k+1} and so on.

case 2: propagation from goaled sub-trees to claused sub-trees see figure 7

The sub-trees t'_j and t'_{j+1} depend on t_k.

case 3 : propagation between goaled and claused sub-trees

The dependencies between the sub-trees are :
- t_{k+1}, t_{k+2}, t'_j, t'_{j+1}, ... , $t"_m$, $t"_{m+1}$ depend on t_k
- $t"_{m+1}$ depends on t_k and $t"_m$.

5.3 The control attribute dependencies

From the different cases given above, we deduce the dependencies between the control attributes on the production of the universal grammar. We recall that we consider one synthesised attribute l_{in} and another inherited attribute l_{out}. We note also that we will give the equality equations between attributes are only represented by edges on the graphs associated with the productions while the others equations are given explicitly and on the graphs see figure 9.

All the attributes equations between the control attributes (l_{in} and l_{out}) can be added to those defining the attributes $\beta, eta, ich, ancandpredicate$ of the universal attribute grammar of [[2]], and integrated in the attribute evaluation algorithm $Eval$ used in the abstract interpretation algorithm. Now, how to use this control attribute in order to avoid the reevaluation of useless sub-trees of θ_i during evaluation process of attributes of θ_{i+1}?.

5.4 Optimized Abstract Interpretation Algorithm

In this algorithm, we distinguish two sub-trees of attributes, the set of attributes for abstract interpretation $B = \{\beta_{in}, \beta_{out}, eta_{in}, eta_{out}, inch_{in}, ich_{out}, predicate\}$ and the set of control attributes $L = \{l_{in}, l_{out}\}$. The strategy of combining tree development and attribute evaluation is the following:
- evaluate the attributes B of the tree θ_i
- develop the tree θ_i from left to right
- for each "greffed" tree development (τ), evaluate the attributes of L and propagate incrementally their values through the tree θ_{i+1}.

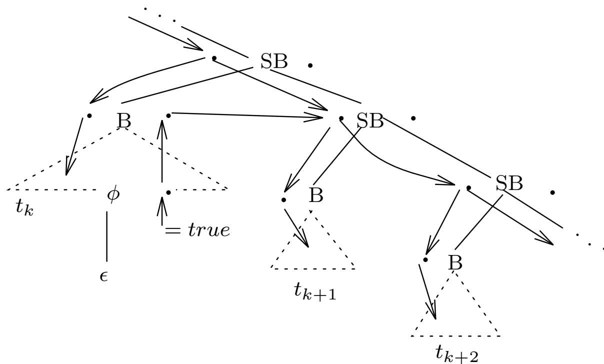

Figure 6: Propagation: goal sub-tree

We give the details of the algorithm.

Procedure Abstract_Interpretation(p : predicate, β : abstract substitution)

```
{i = 0; /* θ0 is the initial tree (see ...... ) */
 initialize the inherited attributes of the root of θ0 :
        βin(root(θ0)) = β;
        etain(root(θ0)) =∅;
 Eval_lr(θ0, root(θ0), {lin}, {lout})
 Loop_Evaluate_&_Develop();
}
```

Procedure Loop_Evaluate_&_Develop()

```
/* This procedure constructs the sequence of increasing trees θi */

{loop
  Optimized_Eval_lr(θi, root);
  if etain(root(θi)) = etaout(θi))
     then {/* θi is the fixed point */
        print(βout(root(θi));
        exit;
       }
     else { /* initialisation of the tree θi+1 */
        θi+1 = θi
        /* construction by steps of the tree θi+1 */
        let U = {u1, ... , un / ∀ i, ui.production = r3
              and ∀ i ∈ [1..n − 1], ui is on the left
          of  ui+1 in θi};
        for u = u1 to un do
           { old_value=lout(u);
```

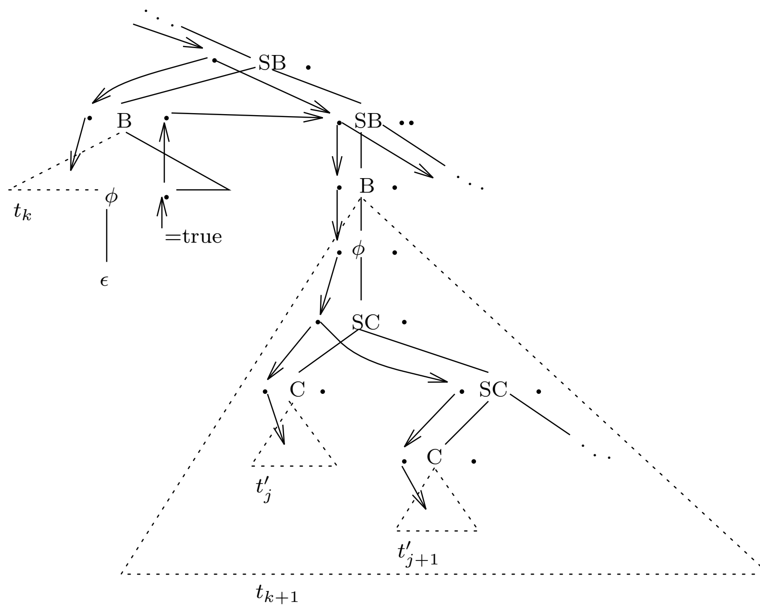

Figure 7: Propagation: goal sub-tree to clause sub-tree

```
        develop_node(u, θ_{i+1});
      /* evaluation of l_in and l_out in the tree
      greffed at u in θ_{i+1} */
        if l_in(u) = true then
          feval_lr(θ_{i+1},u,{l_in},{l_out})
          propagation of l_in and l_out outside the
          tee greffed at u in θ_{i+1} */
            if l_out(u) ≠ old_value then
              Propagate_Incremental_lr(u);
    }
    eta_in(root(θ_{i+1})) = eta_out(root(θ_i))
      i = i+1;
  }
endloop;
}
```

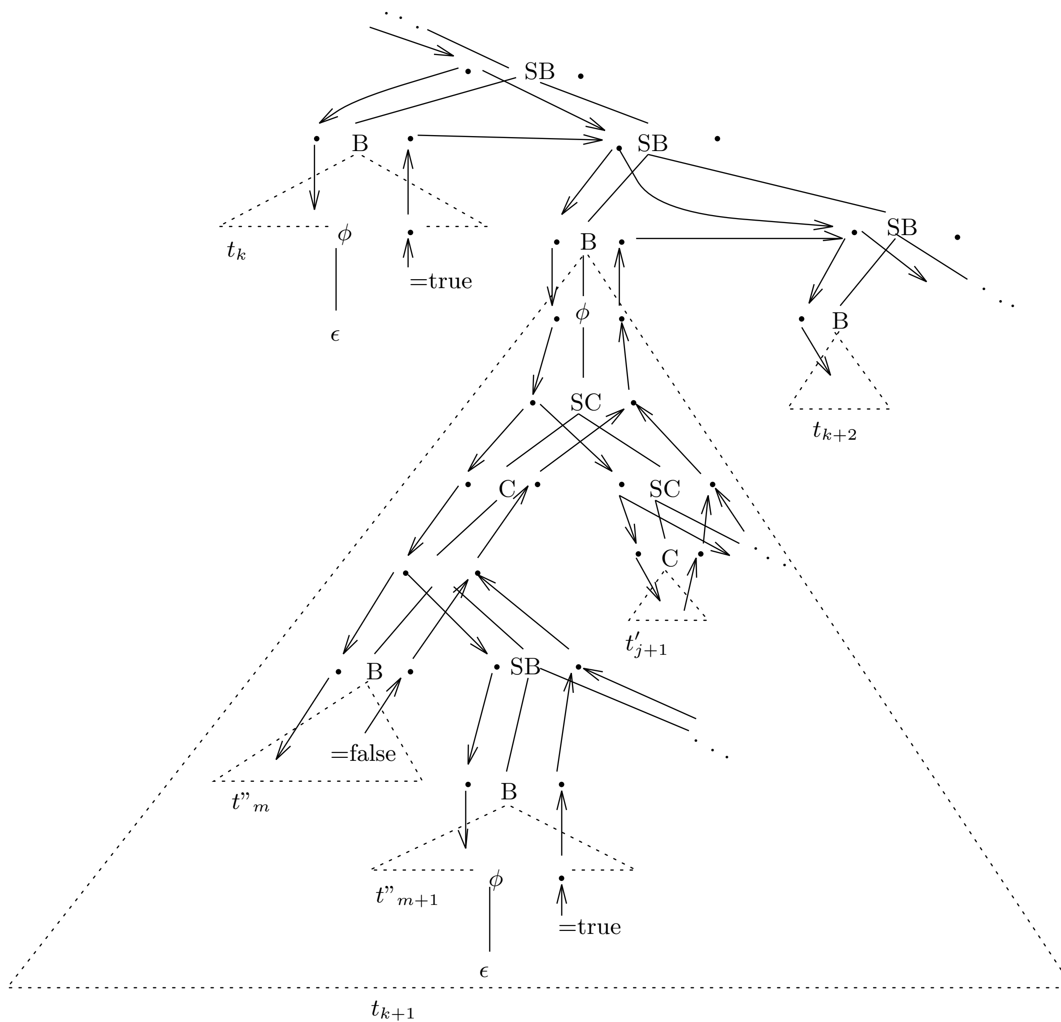

Figure 8: Propagation: goal and clause

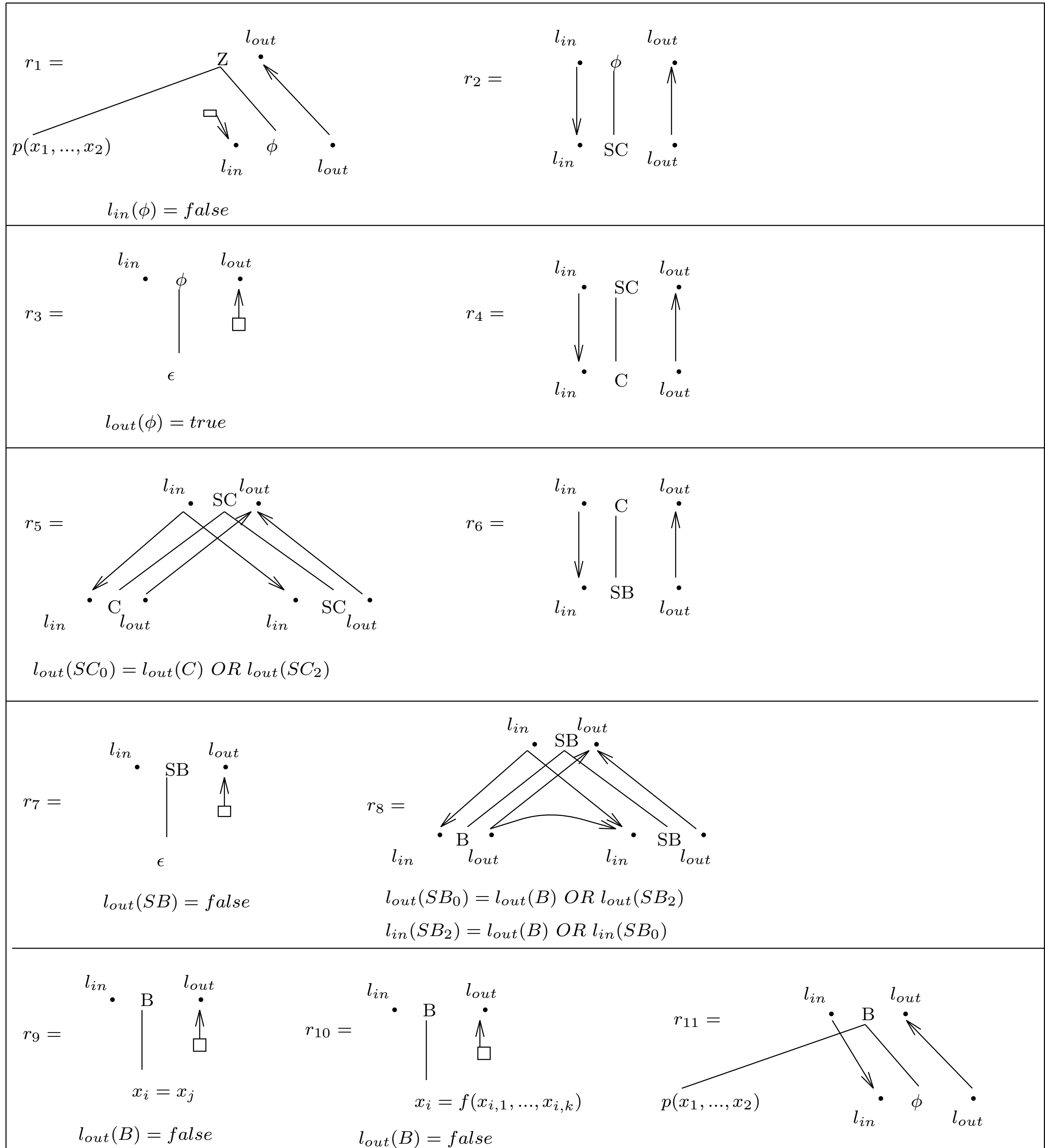

Figure 9: Control attribute dependencies

Procedure Optimized_Eval_lr(θ : in tree, u : in node)
/* This procedure evaluates the attributes of B in the not useless sub-trees */

```
{ let u_1, u_2, ..., u_n be the sons of u in t from left to right;
  for v = u_1 to u_n do
   { if l_in(v)= true or l_out(v) =true
      then
        { calculate the inherited attributes
            {β_in, eta_in, ich_in, anc, predicate}
            at the node v;
          Optimized_Eval_lr(θ,v);
          calculate the synthesized attributes
           {β_out, eta_out, ich_out} at the node v;
        }
   }
}
```

Procedure Propagate_Incremental_lr(θ : tree , u : node)
/* This procedure evaluates the attributes of l_{in} and l_{out} while It is needed */

```
{ if (u ≠ nil) then
   { Eval_lr(u,{l_in},{l_out});
      /* evaluation of the sub-tree at u */
    while (u ≠ root) & (brother(u) = nil) do
      { calculate (l_out(father(u)));
       u = father(u);
      }
      if u ≠ root then
      { old_value = l_in(brother(u));
        calculate(l_in(brother(u)));
        if l_in(brother(u)) ≠ old_value then
         Propagate_Incremental_lr(θ,brother(u));
      }
   }
}
```

6 Conclusion

Our method to compute the abstract semantics of logic programs use attribute evaluators on a sequence of increasing trees. The principle consists of constructing a sequence of finite attributed trees that converges to a fixed point. Obviously, the fixed point is an attributed tree such that its successor is the same tree with the same attribute values. As some attributes of subtrees, cannot be changed, after their first evaluation, we use an incremental algorithm which prunes some branches of tree and develop only the subtree which can improve the *"oracle"* (i.e. set of abstract tuple) *Eta*. Successful implementation of this algorithm yielded memory storage saving and run-time reduced. We hope that our method can be applied for verifying more general concurrent systems such as circuits and network protocols in case of the constraint domain over infinite trees.

References

[1] **K. Barbar, K. Musumbu** *Expressing abstract interpretation of PROLOG by attribute grammars*, rapport interne n^o $93 - 9$, Univ Bordeaux-1, Mars 1993.

[2] **K. Barbar, K. Musumbu** *Implementation of Abstract Interpretation Algorithms by Means of Attribute Grammar*, IEEE,SSST,March 1994,p 87-93.

[2] **M. Bruynooghe et al.** *Abstract interpretation: Towards the global optimization of logic programs*, In Proc. 1987 Symp. on Logic Programming, IEEE Society Press, pp 192-204.

[3] **M. Bruynooghe**

A practical framework for the abstract interpretation of logic programs, Journal of Logic Programming, 1991.

[4] **Corsini, M-M., Le Charlier, B. , Musumbu, K. and Rauzy, A** *Efficient Abstract Interpretation of Prolog Programs by means of Constraint Solving over Finite Domains (Extended Abstract)*, Proceedings of the 5th Int. Symposium on Programming Language Implementation and Logic Programming, PLILP93,Tallinn, (Estonia),1993.

[4] **B. Courcelle, P. Deransart**

Proofs of Partial Correctness for Attribute Grammars with Applications to Recursive Procedures and Logic Programming,Academic Press, 1988.

[5] **P. Cousot, R. Cousot** *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints*, POPL 1977, Sigact Sigplan, pp 238–252.

[6] **T. Kanamori, T. Kawamura** *Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation*, Technical report, ICOT, 1987.

[7] **B. Le Charlier, K. Musumbu, P. Van Hentenryck** *A generic abstract interpretation algorithm and its complexity analysis*, In Proc ICLP 91, June 91.

[7] **B. Le Charlier and P. Van Hentenryck** *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, TOPLAS, January 94.

[8] **D. Knuth** *Semantics of context-free languages*; Math. Systems Theory 2 (1968), pp 127-145, 5th ICLP–SLP 88; tutorial N°2.

[9] **J.W. Lloyd** *Foundations of Logic Programming*, Springer Verlag, 1987.

[10] **K. Marriott and H. Sondergaard** *Notes for a Tutorial on Abstract Interpretation of Logic Programs*, NACLP 89, Cleveland, 1989.