MASTER IN COMPUTER SCIENCE

Deriving run time properties of Logic Programs by means of Abstract Interpretation An implementation

Rouard, Denis

Award date:
1993

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21
B-5000 NAMUR

Deriving run time properties of Logic Programs by means of Abstract Interpretation:

An implementation

Denis Rouard

Mémoire présenté en vue d'obtenir le grade de
Licencié et Maître en Informatique

Promoteur : Baudouin Le Charlier

Année académique 1992 - 1993

Abstract

A lot of researchs are done at the present time to improve the Logic languages. Those languages possess a lot of advantages, one of the most important is the multidirectionnality. Multidirectionnality permit to write a program which can be used in several manners. The same program can do much more than it first utility. This could be an inconvenient, multidirectionnality forces the compiler to explore a lot of unuseful possible solutions. This is due to the fact that the computer can not differenciate a clever solution from a stupid one. Furthermore, it can be observed that the multidirectionnality is often not really used in practice. One of the main purposes of the abstract interpretation is to improve the "intelligence" of such a compiler.

The static analysis of programs covers all the treatments that could be applied on a program behalve of its execution. The aim of the abstract interpretation (which is a technique of static analysis) is to compute some properties of the output of a program without executing it. One of the basic principle of the abstract interpretation is to execute a program not on its normal domain but on an abstraction of its normal domain (understand an abstraction of its input). The aim of this work was to implement an abstract domain proposed by the Katholiek Universiteit van Leuven. and to run it with an abstract interpreter developped in the Facultés Universitaires Notre Dame de la Paix.

Resumé du mémoire

Beaucoup de recherches sont actuellement en cours dans le but d'améliorer les langages logiques. Ils possèdent beaucoup d'avantages, parmi ceux-ci, un des plus impressionnants est la multidirectionnalité. Une même procédure peut servir à plusieurs usages différents. La multidirectionnalité n'a pas que des avantages, elle pose des problèmes d'efficacité. Elle force le compilateur à explorer toutes les solutions possibles d'une même procédure ce qui ralentit très fort l'exécution du programme. Ceci est du au fait que le compilateur est incapable de différencier une solution intelligente d'une solution stupide à un problème posé. De plus, force est de constater que la multidirectionnalité est peu utilisée en pratique. Le but de l'interprétation abstraite est d'améliorer "l'intelligence" des compilateurs de langages logiques.

L'analyse statique des programmes couvre tous les traitements qui peuvent être appliqués à un programme en dehors de son exécution. Le but de l'interprétation abstraite (qui est une technique d'analyse statique) est de déterminer des propriétés des résultats d'un programme sans exécuter celui-ci. Une des techniques fondamentales utilisée en interprétation abstraite est d'exécuter un programme non pas sur son domaine réel mais sur une abstraction de son domaine (comprenez sur une abstraction des données en entrée). Le but de ce travail était d'implementer un domaine abstrait proposé par Gerda Janssens de l'Université Catholique de Louvain dans un interpréteur abstrait développé aux Facultés Universitaires Notre Dame de la Paix.

Contents

List of Figures

Preface

The static analysis of computer programs is a wide domain of research in which we can find a promising technique called *Abstract Interpretation.* This technique allows us to give properties on the output of Logic Programs without executing it. A lot of universities and teams of researchers are working on this subject all over the world, e.g. the university of Bordeaux, Brown, Leuven, Namur, Padova.

Some implementations have been performed in those places. In particular, Gerda Janssens and Maurice Bruynooghe have developed, in Leuven, a framework for abstract interpretation. This framework was implemented in Prolog and the aim of this work is to mqk a new implementation, this one in the C language.

Firstly, we present the theoretical background underlying the abstract interpretation, as few words of introduction are based in [1]. Then an introduction to the fixpoint theory is based on [12]. And finally, the introduction to normalized Prolog programs have been written on the basis of [14].

In the second chapter, we present in a very intuitive way, the aim of this work. It will be interesting for readers which didn't know the framework developed by Gerda Janssens and Maurice Bruynooghe.

The third and fourth chapters are excerpts from the PhD thesis of Gerda Janssens. Only the proofs have been removed from this copy. They are included in order to make this work reasonably "self-contained" and should be skipped by reader knowing the work of Gerda Janssens.

The two next chapters present all the data structures chosen to match those presented in the thesis. They contain a brief description of the major difficulties encountered during the implementation and an explanation of the solutions. Together, they form the main chapters on this paper.

Chapter 7 describes the abstract operations interfacing with the existing frameworks. It contains a description of those operations on two different frameworks. The first framework was developed by the team of researchers of Maurice Bruynooghe and the others by Baudouin Le Charlier and Pascal van Hentenryck. This work was to adapt the operations of the first framework to fit on the operations required by the second one.

Chapter 8 gives the result of this implementation. The chapter 9 proposes an optimization and gives the conclusions of the work.

During this implementation, we used as the basis of the work the two following articles, the PhD thesis of Gerda Janssens "Deriving run-time properties of Logic programs by means of abstract interpretation" ([7]) and the paper "Efficient and accurate algorithms for the abstract interpretation of Prolog programs" written by B. Le Charlier, K. Musumbu and P. van Hentenryck ([9]).

Acknowledgments

First of all, I would like to thank Gerda Janssens who invited me at the Katholiek Universiteit van Leuven. I thank Baudouin Le Charlier and Vincent Englebert who supported this project in important ways and gave me the opportunity to fulfill my job.

I also thank all those who have contributed to this report, specially Maryse, Frounz and Laurent.

Last but not least, I like to thank my parents, without whom these five years would not have been possible.

Chapter 1

Introduction

1.1. Static analysis and abstract interpretation

The curent programmer's world is composed of several type of languages. Each people which has notion of programming eared to object oriented languages, imperative languages.... We can distinct several generic types of languages among them we can found the Logical Languages which are declarative in the sense that a predicate defines a relation between the variables but gives no idea concerning the way to compute the "transformation" to apply on the input to obtain the output.

1.2. Usefulness of abstract interpretation

A lot of researchs are done at the present time to improve the Logic languages. Those languages possess a lot of advantages, one of the most important is the multidirectionnality but this advantage is in the same time the most inconvenient. Multidirectionnality forces the compiler to explore a lot of unuseful possibilities of solutions. This is due to the fact that the computer can not differenciate a clever solution from a stupid one. Furthermore, it can be observed that the multidirectionnality is not really often used in practice. One of the main purposes of the abstract interpretation is to improve the "intelligence" of such a compiler. It can be done with a detection in the program of the user data which will be really used and thus permits to generate more efficient programs.

The static analysis of programs covers all the treatments that could be applied on a program when not executing. The aim of the abstract interpretation (which is a technique of static analysis) is to compute some properties of the output of a program without having to execute it. One of the basic principle of the abstract interpretation is to execute a program not on its normal domain but on an abstraction of its normal domain (understand an abstraction of its input). This implies that the components of the abstract domain represents useful properties of the normal domain. Furthermore, the abstract interpretation of a program must be performed in a non-infinite time and must be as efficient as possible.

1.3. Mathematical Preliminaries : the fixpoint theory

Before begining the description of the abstract domain implemented in this paper, a theoretical background is needed. We will define in this section the fixpoint theory's principles.

1.3.1. Complete Partial Order

Definition 1.1
Let S be a set. A *relation* R on S is a subset of $S \times S$.
We express the fact that $(x,y) \in R$ by $x\ R\ y$.

Definition 1.2
A *partial order* on a set S is a relation denoted by $\leq$ such that :

- $x \leq x \ \forall x \in S$
- $x \leq y \wedge y \leq x \Rightarrow x = y \ \forall x,y \in S$
- $x \leq y \wedge y \leq z \Rightarrow x \leq z \ \ \forall x,y,z \in S$

Definition 1.3
Let S be a set with a partial order relation $\leq$.
Then $a \in S$ is *an upper bound* of a subset $X \subseteq S$ if $x \leq a \ \forall \ x \in X$. And $b \in S$ is *a lower bound* of X if $b \leq x \ \forall \ x \in X$.

Definition 1.4
Let S be a set with a partial order relation $\leq$.
Then $a \in S$ is *the least upper bound* (noted *lub*) of a subset $X \subseteq S$ if a is *an upper bound* of X and, for all upper bound $a' \in X$ we have $a \leq a'$. Similarly, we can define the *greatest lower bound* (noted *glb*). If they exist, *lub* and *glb* are unique.

Definition 1.5
A *chain* of $(S,\leq)$ is an increasing sequence $(x_i)_{i=0}^{\infty}$ verifying $x_i \leq x_{i+1} \ \forall \ i \geq 0$.

Definition 1.6
We say that $(S,\leq)$ is a complete partial order (cpo) if there is a least element called bottom noted $\perp \in S$, and if all chains of S have a *lub* noted $\coprod_{n=0}^{\infty} x_n$.

1.3.2. Lattice

Definition 1.7
A *lattice* is a partially ordered set $(S,\leq)$ in which any two elements s_1 and s_2 and a *lub* and a *glb* in S.

Definition 1.8
A lattice $(L,\leq)$ is a *complete lattice* if *lub(X)* and a *glb(X)* exist for any subset X or L.
So, we have : $lub(L) = T$, $glb(L) = \perp$.

1.3.3. Monotony

Definition 1.9
Let L_1,L_2 be complete lattices and $T : L_1 \rightarrow L_2$ be a mapping. We say that T is *monotonic* iff

$$x \leq y \text{ in } L_1 \Rightarrow T(x) \leq T(y) \text{ in } L2$$

1.3.4. Continuity

Definition 1.10
Let L be complete lattice and $T : L \rightarrow L$ be a transfomation. We say that T is *continuous* if for every chain $(x_n)_{n \geq 0}$ we have :

$$T(\coprod_{n=0}^{\infty} x_n) = \coprod_{n=0}^{\infty} T(x_n)$$

1.3.5. Fixpoint and fixpoint theorem

Definition 1.11
Let L be complete lattice and $T : L \rightarrow L$ be a mapping. We say that $a \in L$ is the *least fixpoint* of T if a is a fixpoint (i.e. $T(a) = a$) and for all fixpoint b of T, we have $a \leq b$.

Fixpoint theorem
If D is a *cpo* and $f : D \rightarrow D$ is a continuous mapping, then f has a least fixpoint $\mu(f) \in D$.
Moreover, the least fixpoint $\mu(f)$ can be defined by :

$$\mu(f) = \coprod_{n=0}^{\infty} f^n(\perp)$$

$$\textit{where} \left\{ \begin{array}{l} f^0(x) = x \\ f^{n+1}(x) = f(f^n(x)) \end{array} \right\}$$

Continuity is the condition which garantees the existence of a fixpoint. The proof of this theorem follows :
By induction and monotony of f, $\{f^n(\perp)\}$ is a chain, and by means of continuity, we deduce :

$$f\left(\coprod_{n=0}^{\infty} f^n(\perp)\right) = \coprod_{n=0}^{\infty} f^{n+1}(\perp) = \coprod_{n=0}^{\infty} f^n(\perp)$$

So, we have that $\coprod_{n=0}^{\infty} f^n(\perp)$ is a fixpoint of f.

1.4. Prolog

Among the Logical language, one is often employed Prolog. Originally, the abstract domain presented here is defined to be used on Prolog programs, but because of its integration into the algorithm developped in [8], the Prolog programs must be normalized. It causes no difficulties to translate a Prolog program into its normalized version.

Let F_i a set of functors of arity i,
P_i the set of the predicate symbol of arity i.

- a *term* is :
  - either a variable,
  - or a construction of the form $f(t_1,...,t_i)$ where f belongs to F_i and where $t_1,...,t_i$ are terms.
- a *normalized atom* is
  - either a built-in
  - or a procedure call
- a *built-in* is
  - either a construction of the form $X_i = X_j$ $(i \neq j)$,
  - or a construction of the form $X_i = f(X_{j_1},..., X_{j_n})$ where the variables $X_i, X_{j_1},..., X_{j_n}$ are distincts two by two and $f \in F_n$,
- a *procedure call* is a construction of the form $p(X_{j_1},..., X_{j_n})$ where $p \in P_n$ and the variables $X_{j_1},..., X_{j_n}$ are distincts two by two.
- a *normalized clause* has the form $p(X_1,..., X_n) \leftarrow SB$ where $0 \leq n$, $p \in P_n$, and SB is a sequence of normalized atoms which contains only program variables. If m is the number of distinct variables appearing in a clause, those variables are the m first : $X_1,...,X_m$.
  $p(X_j,..., X_j)$ is the *head* of a clause and p its name.
  SB is the body of the clause.
- A *normalized procedure* is a non empty sequence of normalized clauses with the same functor : $C_1,...,C_m$.
- A *normalized program* is a set of normalized procedures with distincts functor such as each functor present in the body of a clause is the functor of a procedure.

Prolog code for ***append***
append([],L,L). append([H|T1],L,[H|T2]) :- append(T1,L,T2).

Normalized version of ***append***
append(X1,X2,X3) :- X1 = [], X3 = X2. append(X1,X2,X3) :- X1 = [X4 | X5], X3 = [X4 | X6], append(X5,X2,X6).

1.5. Concrete and abstract domain

As evoqued before, the abstract interpretation does not work on the real data domain (which is called the concrete domain). It works on an abstract domain which is a simplified image of the concrete domain. For example, the standard computation domain (e.g. the set of integers) is approximated by an abstraction of this domain (e.g. the set of signs {+,-,0,?} representing the set of strictly positive integers (+), the set of strictly negative integers (-), zero (0) and the set of all the integers(?)). The abstract domain is commonly required to be a cpo. The link between the concrete and the abstract semantics is usually given by a concretization function from the abstract domain to the concrete domain. The concretization function specifies which concrete object are represented by an abstract object. In the above example, the concretization of + was the set of strictly positive integer and we denote this information by

$$\gamma(+) = \{X \mid X \text{ is an integer and } X > 0\}$$

The operation which associates a set of concrete objects with an abstract object is called abstraction. This function establishs the link between the abstract and the concrete semantics. This operation is denoted by α.

Those two functions verify the following conditions :

$$\forall c \in C: \gamma(\alpha(c)) \subseteq c$$
$$\forall a \in A: \alpha(\gamma(a)) = a$$

Once the abstract domain is defined, we must introduce the operations on the abstract domain. In order to deduce properties on the result of a program, we need to define the abstract operations for each concrete operation. Each concrete operation which is an operation on the concrete domain is associated with an abstract operation on the abstract object. Still by means of the same example, the multiplication and the addition of integers can be abstracted. We note $\otimes$ the asbstract multiplication operation which is corresponding to the concrete multiplication operation $\times$. We note $\oplus$ the asbstract addition operation which is corresponding to the concrete addition operation +. Convertions array are proposed on both next arrays.

$\otimes$	0	+	-	?
0	0	0	0	0
+	0	+	-	?
-	0	-	+	?
?	0	?	?	?

$\oplus$	0	+	-	?
0	0	+	-	?
+	+	+	?	?
-	-	?	-	?
?	?	?	?	?

1.6. Abstract interpreter

As all the other abstract domains, the abstact domain implemented here is not "stand alone" in the sence that it will be integrated into an abstract interpreter. This abstract domain was conceived to be used by the abstract interpreter presented by [2] and was made at the Katholiek Universiteit van Leuven but will be implemented for the abstract interpreter from the Facultés Universitaires Notre-Dame de la Paix.

Chapter 2

Intuitive overview of the abstract domain

The aim of this chapter is to present unformally the abstract domain implemented in this paper. It is proposed without any concern of rigour and formalism, it is an intutitive overview of the abstract domain to show the usefulness of the domain. A reader used to the components of this domain can pass directly to the next chapter.

An abstract domain is composed of different parts called here *component*, each component bring out precision to the abstract domain in the sense that it permits to give a best approximation of the concrete domain. The abstract domain proposed here is formed by two linked component, the type-graph component and the same-value component.

2.1. Type-graph component

Normalized Prolog programs, like all Prolog programs, are using variables. The type-graph component permits to express the different values a variable can have during the concrete interpretation of the program. For example, suppose that we have the following Prolog program :

$$p(X_1) :- X_1 = a.$$
$$p(X_2) :- X_2 = b.$$

We will sketch a very simplified abstract interpretation on this example. The abstract interpreter parses the program and see that X_1 is unified with the value *a*, so it create a graph to represent this unification, as it can be observed in the next figure.

Figure 2.1 : $X_1 = a$

After it meets the second unification X_2 = b. We can remark that the head of the second clause is the same than the head of the first clause. The variable concerned with the

unification X_2 = b is thus the same (up to renaming) than the variable concerned with the first unification X_1 = b. The abstract interpretation creates thus the type-graph corresponding to the next figure.

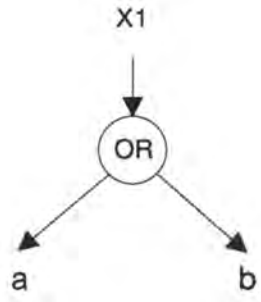


Figure 2.2 : *X_1 = a or X_1 = b*

The graph express the fact that X_1 can have either the value *a* or the value *b*.

This situation is very simple but the type-graph can be created to denote more complex configurations. In the previous example, we consider that X_1 has no input value but it is not always the case, the user can force a value to a variable. For example, if the user wants to observe the transformation of a list, which is a typical data structure of Prolog, he can enter the following type-graph as input.

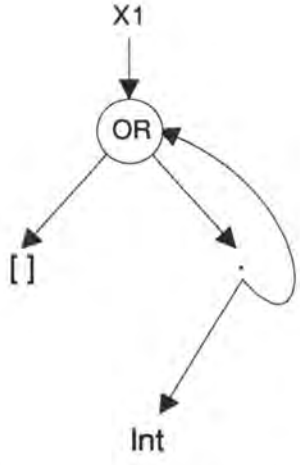


Figure 2.3 : *Type-graph representing a list of integers*

To understand well this type graph, the reader has to remember the notion of list. An integer list is either an empty list [ ] or an integer followed by a list.

<List> ::= = [ ] | . (Integer, <List>)

Those two examples suffice to have a short idea of the type-graph component used in this abstract domain. As expressed before, the domain is composed of two components. The reason of the existence of this second component is more difficult to explain.

2.2. *Same-value component*

Let us suppose that we must perform the abstract interpretation on the following Prolog program and that the abstract domain contains only the type-graphs component.

$$p(X_1,X_2, X_3) :- X_1 = f(X_2), X_3 = X_1, X_3 = f(1).$$

Let us suppose that X_2 denotes an integer. When the abstract interpreter parses this program, it unifies X_1 to $f(X_2)$, after that it unifies X_3 and X_1. So, after this unification we can say that X_3 is equal to $f(X_2)$ and that X_1 is the same. At this moment, we know that X_2 and X_3 have the same value. Afther we make the third goal, it unifies X_3 to $f(1)$. If there is not same value component, we lose the information that X_1 is equal to $f(1)$ too.

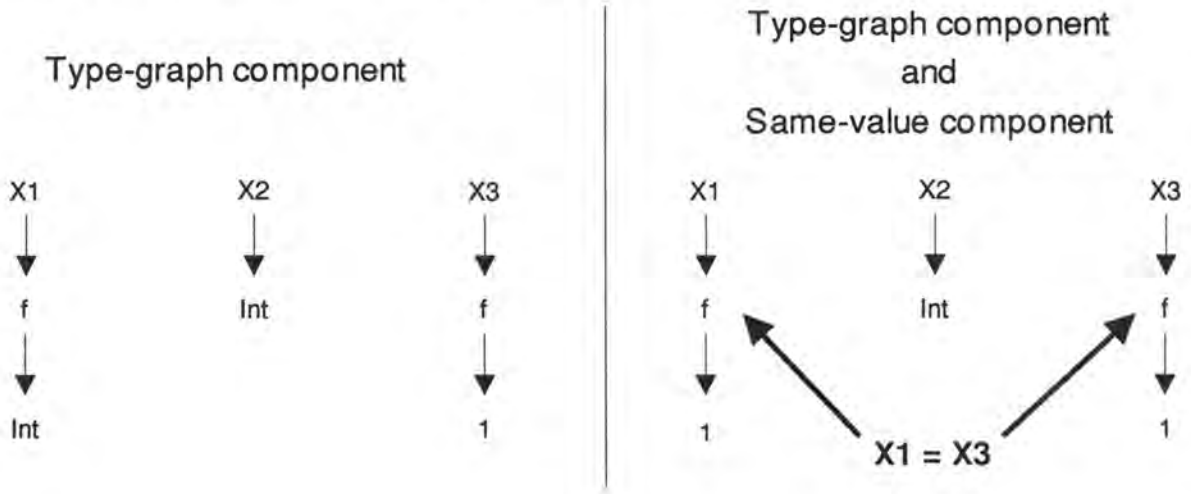


***Figure 2.4** : Usefulness ot the same-value component*

This is the reason of the existence of the same-value component. It allows to keep track of all the unifications done during the abstract interpretation. The fact that X_1 and X_3 have the same value implies that the same term should be used when considering the concrete variables approximated by the abstract variables.

2.3. *Summary*

This short introduction to the domain presents the two components used in the abstract domain described in the two next chapters. We underline the fact that this presentation is incomplete in the sense that it can not be considered that a real abstract interpretation was done on the previous examples. Furthermore this presentation was unformal in the sense that none of the real abstract interpretation operations were presented here. The aim of this chapter was to softly introduce the reader to the components of the domain before formalizing it.

Chapter 3

Rigid Types

This chapter and the following one are excerpts of the chapter 4 and 5 of [7]. Only the proofs have been removed form this copy. The aim of this work is to implement all the operations proposed in those two chapters.

The types are said to be rigid because usually their diversity and their expressive power is of the right level of precision, but they are sometimes a too crude approximation. This chapter defines the type graphs which we use to represent types. The Chapters 4 presents the abstract domain.

3.1. Definition of normal type graphs

3.1.1. Graphs

Types are represented by a special kind of directed graphs. First we introduce the terminology that we adapt from the graph theory.

A *directed graph* G is a pair (N, A) where N is a finite non-empty set, and A is a relation on N (A is any subset of N x N). Each element in N is called a *node*, and each pair in A is called an *arc*. The arc (n, m) *leaves* the node n and *enters* the node m. We say that n is a *predecessor* of m, and m is a *successor* of n. PRED(n) denotes the set of predecessors of node n and SUCC(n) denotes the successors of node n. The *indegree* of a node n is #PRED(n) and the *outdegree* of n is #SUCC(n), where #S denotes the cardinality of a set S. The arcs entering a node n are called the *incoming* arcs of n, and the arcs leaving a node n are called the *outgoing* arcs of n. A *path* is a finite sequence of one or more nodes such that, if the sequence contains two or more nodes and is denoted by $(n_1, ..., n_k)$ with $k \geq 2$ then $(n_i, n_{i+1}) \in A$ for $1 \leq i \leq k - 1$. A *simple* path is a path with distinct nodes. If $k \geq 2$, then $(n_1, ..., n_k)$ is a path *from* n_1 *to* n_k with *length* $k - 1$, and n_k is said to be *accessible* from n_1. As a special case, a single node denotes a path of length 0 from itself to itself. A *cycle* is a path $(n_1, ..., n_k)$ where $n_1 = n_k$. A *simple* cycle is an arc from a node to itself. A path is *acyclic* or *non-circular* if it does not contain a cycle.

A directed acyclic graph (acronym DAG) is a directed graph without any cycles. A (rooted) *tree* is a DAG satisfying the following properties:

1. There is exactly one node, called the *root*, with indegree 0.
2. Every node except the root has indegree 1.
3. There is a path from the root to each node.

If (u, v) is an arc in the tree, then u is called the *parent* of v and v is called a *son* of u. The *ancestor* and *descendant* relations are the reflexive and transitive closures of the respective parent and son relations. Node n is called a *proper* ancestor (descendant) of node m iff n is ancestor (descendant) of m and $n \neq m$. A *leaf* is a node n with outdegree 0.

3.1.2. Type graphs

A *type graph* T is a triple (n, A_F, A_B) where $T_r = (N, A_F)$ is a rooted tree whose arcs in A_F are called *forward* arcs and A_B is a restricted class of arcs, *backward* arcs, superimposed on T_r . T_r is called the *underlying* rooted tree of the type graph T. In a type graph T, ANC^* (n) (respectively $ANC^+(n)$) is the set of ancestors (respectively proper ancestors) of the node n in T_r. The bacward arcs (n, m) in A_B have the property that $m \in ANC^*(n)$. $FDESC^*(n)$ (respectively $FDESC^+(n)$) is the set of descendants (respectively proper descendants) of the node n in T_r. $DESC^*(n)$ (respectively $DESC^+(n)$) is the set of descendants (respectively proper descendants) of the node n in T.

Note that $DESC^*(n) = \{n\} \cup DESC^+(n)$ (idem for FDESC and ANC). Note that the set of forward descendants, FDESC, only takes into account the arcs A_F, whereas the set of descendants, DESC, takes into account $A_F \cup A_B$. A forward path is a path composed from forward arcs.

Each node n of a type graph has a label, denoted by *lb(n)* indicating the kind of term it describes and the nodes are divided into three classes:

- *simple nodes* have a label from the set {max, ⊥, Int, Real, ...} and their outdegree is 0. We give some of them a specific name: max-node, ⊥-node.
- *functor nodes* are labeled with a functor f/k and have outdegree k with $k \geq 0$ (a constant has arity 0).
- *Or-nodes* have the label OR and have outdegree k with $k \geq 0$.

We use the convention that *n/i* denotes the i^{th} son of node n and the set of sons of a node n is then denoted as {n/1, ... , n/k} with k = outdegree(n).

Definition 3.1.

A type graph T (N, A_F, A_B) has the following characteristics:

P_1. There is exactly one node, called the root n_0, with no incoming forward arc.
P_2. All the nodes, except the root, have exactly one incoming forward arc.
P_3. There is a unique path from the root to each node only taking into account the forward arcs.
P_4. For each backward arc $(n, m) \in A_B$, $m \in ANC^*(n)$.
P_5. Simple nodes are leaves, functor nodes labeled f/k have outdegree k and the outdegree of OR-nodes is ≥ 0.

We asume that type graphs have disjoint sets of nodes (node names are unique over all the type graphs).

The type graphs in Definition 3.1. are called *rigid types*.

The graphical representation of the type graphs is straightforward. The nodes of a type graph are represented by their label, only the OR-node is encircled as we often drop its label. The direction of the arc is indicated by its arrow: forward arcs are drawn downwards, backward arcs upwards. The root of the type graph is the topmost node. Examples are shown in Figure 3.1.

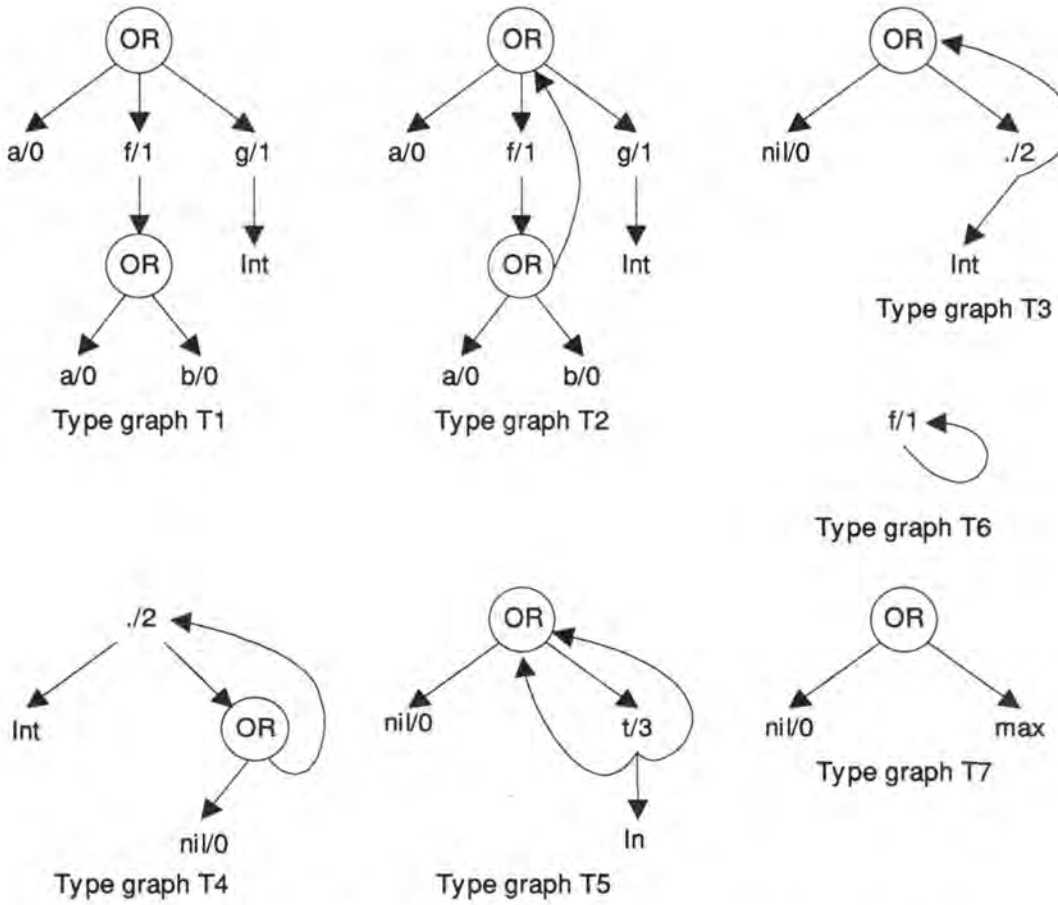


***Figure 3.1** : The graphical representations of some type graphs*

The information described by a type graph can also be expressed by a context free grammar with a set of nonterminal symbols {T, T_1, T_2, ... } called type names where T, the name of the type to be defined is the start symbol and corresponds to the root of the type graph.

For the type graphs T_2, T_3, T_4 and T_5 of Figure 3.1, we can write:

T_2 := a | f(a | b | T_2) | g(Int)

T_3 := nil | .(Int, T_3)

T_4 := .(Int, nil | T_4)

T_5 := nil | t(T_5, Int,T_5)

It is convenient to use more expressive type names such as List instead of T_3, ListOne instead of T_4 and Tree instead of T_5. It is rather straightforward to divise algorithms to extract a context free grammar from a type graph and to construct a type graph from a context free grammar. Context free grammars are used for the communication of types as they are more compact than the graphical representation of the type graphs. See also [Mis84a] for similar ideas. We prefer to use the type graphs in the following definitions and algorithms because existing terminology (such as nodes, arcs, ancestors,...) can be used to refer to components of the rigid types.

3.1.3. Denotation of a type graph

Let S_V be a set of variables and S_{max} the set of all the terms constructed with the functors and the constants in the program together with the variables from S_V and the terms in the primitive types (e.g. Int, Real, ...). We assume that each primitive type, P, represents a set of ground terms with depth one, S_P, and that these sets are mutually disjoint. The set of finite terms represented by a node n in the type graph T is said to be the *denotation* of the node n, ID(n).

Definition 3.2.
The denotation of a node n in a type graph, ID(n):
if lb(n) = max then $ID(n) = S_{max}$
else if $lb(n) = \bot$ then $ID(n) = \varnothing$
else if lb(n) is a primitive type P then $ID(n) = S_P$
else if (lb(n) = f/k and n/1, ... , n/k are its sons)
then $ID(n) = \{f(t_1, \ldots, t_k) \mid t_i \text{ is finite and } t_i \in ID(n/i) \text{ for } 1 \leq i \leq k\}$

else $ID(n) = \bigcup_{i=1}^{k} ID(n/i)$ as lb(n) = OR and n/1, ... , n/k are its sons

Note that the order of the sons of a functor node is important because they correspond to the arguments, whereas the order of the sons of an OR-node is irrelevant. Observe that if the condition "t_i is finite" were dropped from the rules for lb(n) = f/k, then infinite terms could be included in the set due to backward arcs. ID(n) can be $\varnothing$ or a (in)finite set of finite terms. With n_0 the root of type graph T, we use ID(T) as a synonym for $ID(n_0)$.
The denotations of the examples in Fig.3.1. are:

$ID(T_1) = \{a, f(a), f(b), g(1), g(2), \ldots\}$
$ID(T_2) = ID(T_1 \cup \{f(f(a)), f(f(b)), f(g(1)), f(g(2)), f(f(f(a))), \ldots\}$
$ID(T_3)$ is the set of all lists of integers.
$ID(T_4)$ is the set of all lists of integers that contain at least one element.
$ID(T_5)$ is the set of all binary trees with integer elements.
$ID(T_6) = \varnothing$
$ID(T_7) = S_{max}$

Proposition 3.1.: *The denotation of a rigid type is closed under substitution.*

3.1.4. Compact type graphs

A same set of finite PROLOG terms (even the empty set) can be represented by a number of different type graphs. This makes the $\leq$-operation (needed during abstract interpretation) quite complex and inefficient. In order to reduce this variety of type graphs, additional restrictions are imposed. In a first step, *compact* type graphs are defined. The expressive power of type graphs is preserved under this restriction.

First we introduce the boolean function *non-empty*-ID. Its value non-empty-ID(n) is true if ID(n) is a non-empty set of finite terms. It can be recursively defined as follows:

Definition 3.3.
non-empty-ID(n) iff
if (lb(n) = f/k and k>0) then $\forall$ i $\in$ [1, k]: non-empty-ID(n/i)
else if lb(n) = OR then $\exists$ i: non-empty-ID(n/i)
else lb(n) $\neq \perp$ % n is a simple node or a 0-ary functor node %[1]

The computation performed by the algorithm implementing Definition 3.3., will be bottom-up, starting from the leaves in DESC(n).

Definition 3.4.
A compact type graph T_c (N_c, A_{Fc}), A_{Bc}) has the following properties:
C_1. T_c is a type graph (with characteristics P_1, P_2, P_3, P_4, and P_5).
C_2. $\forall$ n $\in$ N_c : non-empty-ID(n) or n is the root.
C_3. $\forall$ n $\in$ N_c : if lb(n) = OR
then outdegree(n) > 1 and not ($\exists$ i : lb(n/i) = max).
C_4. $\forall$(n, m) $\in$ A_{Fc} : if lb(n) = lb(m) = OR then indegree(m) > 1.
C5. $\forall$(n, m) $\in$ A_{Bc} : n $\neq$ m and the forward path (m, ..., n)
contains at least one functor node.

Algorithm 3.1.: T_c=compact(T)

Perform one of the following transformation steps on T until no step is applicable:

1. If non-empty-ID(n) does not hold for a functor node n, then it is replaced by a $\perp$-node (This includes the case that a $\perp$-node is the son of a functor node). The outgoing arcs of n, all nodes m $\in$ FDESC$^+$(n) and the outgoing arcs of all such m are removed.

2. If an OR-node has a $\perp$-node as son, then that son and the arc to it are removed.

3. If n is a OR-node with an arc (n, n) , then the arc (n, n) is removed.

4. If an OR-node n has no sons, then n is replaced by a $\perp$-node.

[1] Note that a comment is added as text between the two %'s.

5. If an OR-node n has a max-node as son, then it is replaced by a max-node. The outgoing arcs of n, all nodes m ∈ $FDESC^+(n)$ and the outgoing arcs of all such m are removed.

6. If a forward arc connects two OR-nodes (n, m) and indegree(m) = 1 , then all arcs (m, k) are replaced by arcs (n, k) and the arc (n, m) and the node m are removed.

7. If an OR-node m has only one son m, then all the arcs (k, n) are replaced by arcs (k, m) and the arc (n, m) and the node n are removed.

8. If there is a backward arc (n, m) and the forward path m, k_1, ..., k_n with k_n = n consists of only OR-nodes then all backward arcs (l, k_i) are replaced by the backward arcs (l, m). (So indegree (k_i) becomes 1 and step 6 is applicable).

The termination of Algorithm compact is obvious as we never extend the type graph but only remove nodes and arcs from it, or replace node labels by ⊥ or max.

Proposition 3.2.: *If T_c is the compact type graph derived from the type graph T by algorithm 3.1., then* ID(T_c) =ID(T).

Proposition 3.3.: *Each cycle in a compact type graph T_c contains at least one OR-node and at least one functor node.*

We give some examples of (the computation of) compact type graphs.

The type graphs T_1, T_2, T_3, T_4 and T_5 of Fig.3.1. are compact.

compact (T_6) results in a ⊥-node by application of step 1.

compact (T_7) applies rule 5 and changes the type graph into a max-node.

Consider T_8 represented in Figure 3.2.a. The node names are given by the superscripts in italic added to the node labels in the type graphs. The backwards arc (n_3, n_3) is removed by step 3 and then step 6 is applicable on arc (n_1, n_3) has indegree(n_3) has become 1. The type graph at this stage is found in Figure 3.2.b. Then there is a backward arc (n_5, n_1) and a forward path (n_1, n_5) consisting of only OR-nodes such that steps 8 and 6 are performed. Figure 3.2.c. gives the compacted type graph, compatc (T_8). Note that the denotation of all the type graphs in Figure 3.2. is the same.

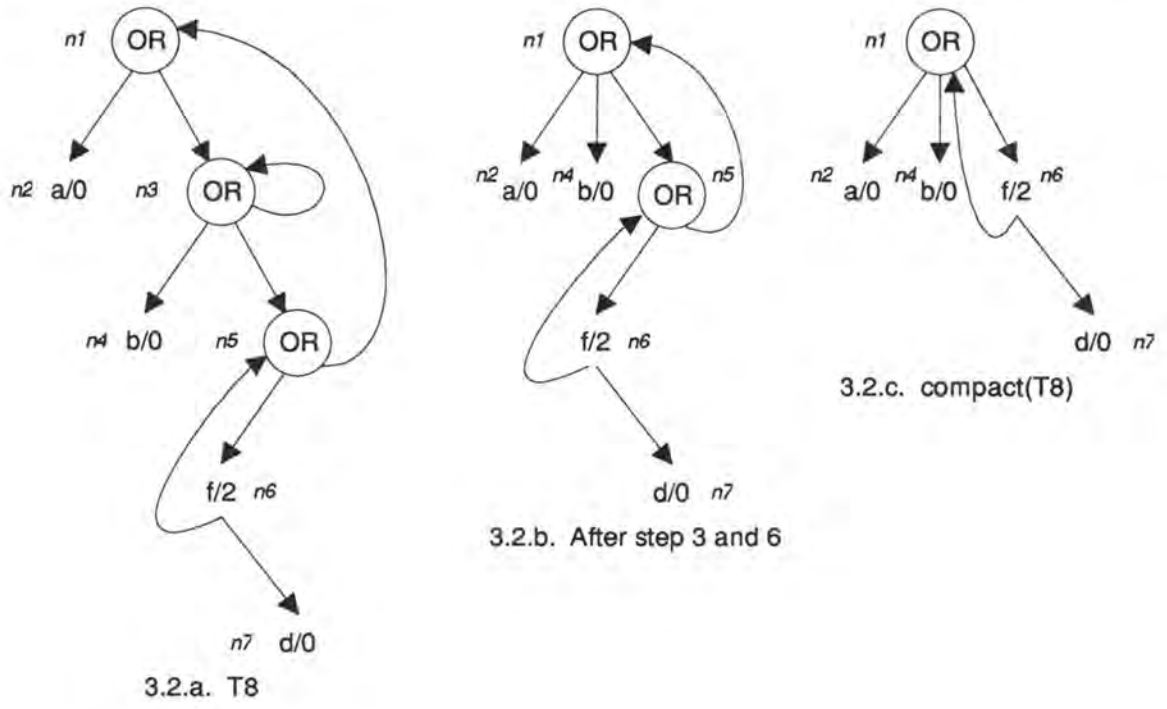


***Figure 3.2** : Example of a compact type graph computed by algorithm 3.1.*

Notice that compact type graphs are not the most economical representation. Nodes in different branches can have the same denotation. In particular, different sons of an OR-node may have overlapping, even identical denotations. This makes testing whether a particular term is in the denotation of a compact type graph and the comparison of the denotations of two type graphs inefficient, so we impose further restrictions.

3.1.5. Normal and restricted type graphs

The restrictions introduced here limit the expressiveness of the type graphs but are necessary to achieve efficient operations and to construct a finite abstract domain (see Chapter 4).

First we introduce the functions *prnd* and *prlb*. The function prnd(n) denotes the set of *principal nodes* of a node n and prlb(n) its set of *principal labels*.

Definition 3.5.

$$\text{prnd}(n) = \text{if } \text{lb}(n) = \text{OR then } \bigcup_{i=1}^{k} \text{prnd}(n/i) \text{ with } k = \text{outdegree}(n)$$

$$\text{else } \{n\}$$

$$\text{prnd}(n) = \{\text{lb}(n_i) \mid n_i \in \text{prnd}(n)\}$$

The compactness of the type graph assures us that

if $\text{lb}(n) \neq \max$ then $\max \notin \text{prlb}(n)$.

Two sets of principal labels are *overlapping* if their intersection is non-empty.

Definition 3.6.
The *principal label restriction* states that each pair of sons of an OR-node must have non-overlapping sets of principal labels.
Normal type graphs are compact type graphs satisfying the principal label restriction.

Definition 3.7.
The *depth restriction* states that the number of occurrences of the same functor on each path of forward arcs in a type graph must be bounded by a constant. *Restricted* type graphs are normal type graphs satisfying the depth restriction.

The compact type graph T_1, T_2, T_3, T_4 and T_5 in Fig.3.1. do not violate these restrictions, neither does compact(T_8). Type graph T_9 of Fig.3.3. violates the principal functor restriction at OR-node n_0 as it has two sons with ./2 as principal label, namely n_2 and n_3. If the allowed depth for ./2 is one, then also the depth restriction is violated at node n_7. The corresponding restricted type graph will be given in Fig.3.7.

Both restrictions limit the expressiveness of type graphs: type graphs violating these restrictions sometimes have to be replaced by a type graph denoting a larger set of terms. Note that both restrictions impose structural properties on the type graphs, which restrict their shape. The same set of terms can be represented by different restricted type graphs as we still allow two nodes in a type graph to represent the same set of terms. So no canonical form is defined for a type graph.

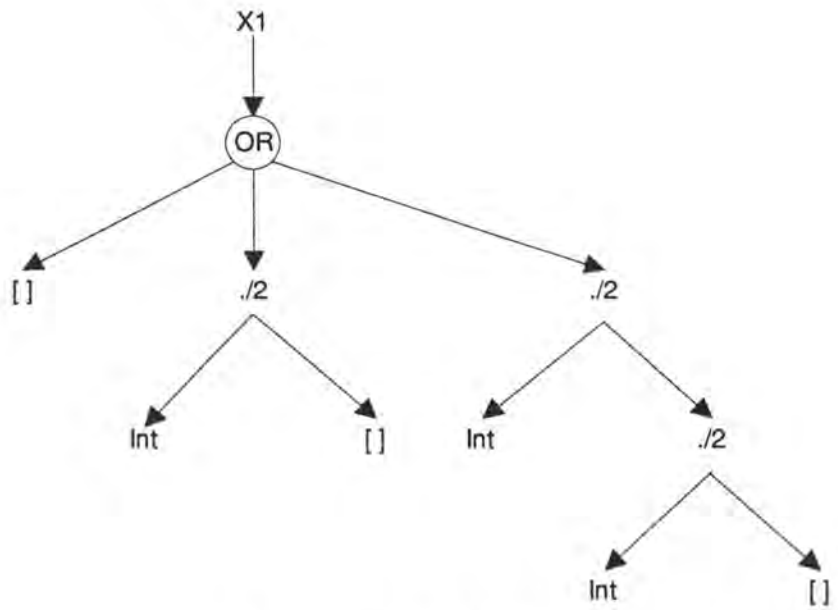


***Figure 3.3** : Type graph T_9*

The algorithm that computes for a given type graph the corresponding restricted type graph, is a special case of the more general algorithm in which, given an old type graph T, a new type graph T' must be constructed which has some specific properties concerning its denotation and also some structural properties. We first discuss the general case. Our strategy to deal with this kind of problems is to leave the old type graph unchanged and to construct stepwise a new type graph (one could say node by node).

The initialization creates the root m_0 of T' with the required denotation which is defined in terms of nodes of T. At this point the root m_0 is called an *unexpanded leaf.* We define a function *fn* which, at every step in the construction of T', associates a set of nodes from the original type graph T to each node in T'. Associated to *fn* we define a second function on the nodes of T', ID-*fn*, which specifies for each newly constructed node m of T' the intended denotation for m. This intended denotation will be determined by the denotation of the nodes in *fn(m).*

Each step extends T' respecting the structural requirements and without decreasing the denotation of its nodes. This is done by transforming one of the unexpanded leaves m of T' into un usual node (after the transformation, m is called a *safe node)* and new unexpanded leaves may be added as sons of m. The nodes of T', in each step of its construction, either belong to S^{ul} which is the set of the unexpanded leaves or to S^{sn} which is the set of the safe nodes.

The case at hand is the Algorithm restrict(n) with n a node (not necessarily the root) in the given compact type graph. It computes the correponding restricted type graph, T', which satisfies the principal functor restriction and the depth restriction, and whose denotation is at least as large as the one of node n. We use the convention that the names of the nodes in T respectively T' are denoted by n respectively m with or without subscripts.

A violation of the principal label restriction at an OR-node n is removed by rearranging its descendants in the new type graph T'. Only the sons involved in the violation are affected. More precisely a node n/j with lb(n/j) = OR and overlapping with other sons of n, is replaced by its sons. Repeating this yields a situation where two sons overlap only when they have the same label and they are both simple nodes or functor nodes. Nodes with the same label are then merged. Or-nodes n/j which did not overlap are preserved and this minimizes the loss of precision.

A functor node violating the depth restriction is added as an alternative to one of its ancestors having that functor as one of its principal labels. The choice of the ancestor affects the loss of precision. There is not always a unique best choice.

The function defined on the nodes in T' for Algorithm restrict (the concrete counterpart of the function *fn*) is *nd*. Its value nd(m) is a set of nodes from T such that we can roughly say that m corresponds to the "union" of all the nodes in *nd(m).* The intended denotation of a node m in T' is given by ID-*nd(m).*

Definition 3.8.

$$\text{ID-nd(m)} = \bigcup_{n \in nd(m)} \text{ID(n)}$$

For the necessary bookkeeping, a second set *nfr(m)* is associated with each node m in T' with $nfr(m) \subseteq nd(m)$. The set *nfr(m)* is a *front* of nodes in the set *nd(m).* We call them *nominees* and require that they satisfy the condition:

Proposition 3.4:

If lb(m) = OR then $\bigcup_{n \in nd(m)} ID(n) = \bigcup_{n \in nfr(m)} ID(n)$ *else nfr(m) = ∅* .

The algorithm uses some additional functions and procedures such as the boolean function *involved-overlap(m, n)* defined for OR-nodes m. It returns true if m has a nominee n which is an OR-node and which overlaps with another nominee of m.

Definition 3.9.
involved-overlap(m, n) iff

$n \in nfr(m)$ and lb(n) = OR and $prlb(n) \cap \bigcup_{k \in nfr(m) \setminus \{n\}} prlb(k) \neq \emptyset$

The boolean function *safe-anc(m, fn, m_g)* is defined for an unexpanded leaf m, a problem dependent function *fn* and a safe node m_g. It returns true if m_g is an ancestor of m, if both nodes have the same *fn*-value and if not all the nodes on the path from m_g to the parent of m are OR-nodes.

Definition 3.10.
safe-anc(m, fn, m_g) iff
$m_g \in ANC^+(m)$ and $fn(m_g) = fn(m)$ and
$\exists\, m_f \in FDESC^*(m_g) \cap ANC^+(m) : lb(m_f) \neq OR$

The procedure *ul-barc(m, m_g)* is defined for an unexpanded leaf m and a safe node m_g. It changes the forward arc to m into a backward arc to m_g.

Definition 3.11.
ul-barc(m, m_g) ≡
$S^{ul} \leftarrow S^{ul} \setminus \{m\}$,
the forward arc (m_p, m) is replaced by a backward arc (m_p, m_g)

Algorithm 3.2.: T' = restrict(n)

Initialization
T' is intialized with a root named m_0.
$lb(m_0) \leftarrow lb(n)$, $S^{ul} \leftarrow \{m_0\}$, $S^{sn} \leftarrow \emptyset$,
if lb(n) = OR then $nfr(m_0) \leftarrow \{n/1, \ldots, n/k\}$ with k = outdegree(n)
else $nfr(m_0) \leftarrow \emptyset$,
$nd(m_0) \leftarrow \{n\} \cup nfr(m_0)$

Repeat
Select a m from S^{ul} and
1. If $\exists\, m_g \in S^{sn}$: safe-anc(m, nd, m_g) then ul-barc(m, m_g)
% see Fig.3.4 %
2. else if (m is a simple node or a functor node with label f/o)
then $S^{ul} \leftarrow S^{ul} \setminus \{m\}$, $S^{sn} \leftarrow S^{sn} \cup \{m\}$

3. else if lb(m) = OR % see Fig.3.5 %
then while $\exists$ n $\in$ nfr(m) : involved-overlap(m, n)
do nfr(m) $\leftarrow$ nfr(m) $\setminus$ {n} $\cup$ ({n/1, ..., n/k} $\setminus$ nd(m))
with k = outdegree(n)
nd(m) $\leftarrow$ nd(m) $\cup$ {n/1, ..., n/k}
od
3.a. if $\exists$ $m_g \in S^{sn}$: safe-anc(m, nd, m_g) then ul-barc(m, m_g)
3.b. else for each n $\in$ nfr(m) : lb(n) = OR
do add a node m/i with lb(m/i) = OR as son to m,
nfr(m/i) $\leftarrow$ {n/1, ..., n/k},
nd(m/i) $\leftarrow$ nfr(m/i) $\cup$ {n}
od
for each label l: ($\exists$ $n_p \in$ nfr(m): lb(n_p) = l and l $\neq$ OR)
do construct the set $\{n_p, \ldots, n_q\} \subseteq$ nfr(m) of all the nodes with the label l,
add a node m/i with lb(m/i) = l as son to m,
nd(m/i) $\leftarrow$ $\{n_p, \ldots, n_q\}$, nfr(m/i) $\leftarrow \varnothing$
od
$S^{ul} \leftarrow S^{ul} \setminus \{m\} \cup \{..., m/i, ...\}$, $S^{sn} \leftarrow S^{sn} \cup \{m\}$

4. else % lb(m) = f/k with k > 0 %
if (adding a functor node for f/k to thet path in T' from the root to m, violates the depth restriction for f/k)
4.a. then % add m as an alternative to one of its predecessors %
4.a.1. if $\exists$ $m_g \in S^{sn}$:
($m_g \in ANC^+(m)$ and lb(m_g) = f/k and n/d(m) $\subseteq$ nd(m_g))
then ul-barc(m, m_g)
4.a.2. else select a $m_g \in S^{sn}$:
($m_g \in ANC^+(m)$ and f/k $\in$ prlb(m_g) and
$\exists$ $m_f \in FDESC^*(m_g) \cap ANC^+(m)$: lb(m_f) $\neq$ OR)
% this existence condition guarantees that m itself is not the cause of f/k $\in$ prlb(m_g). In this case nd(m) $\subseteq$ nf(m_g) is not necessarily valid and we preferably select m_g with nd(m) $\cap$ nd(m_g) as large as possible. %
$S^{ul} \leftarrow (S^{ul} \setminus FDESC^+(m_g)) \cup \{m_g\}$,
$S^{sn} \leftarrow S^{sn} \setminus FDESC^*(m_g)$,
% the nodes of $FDESC^+(m_g)$ and their arcs are removed %
if lb(m_g) = OR
then nfr(m_g) $\leftarrow$ nfr(m_g) $\cup$ nd(m),
nd(m_g) $\leftarrow$ nd(m_g) $\cup$ nd(m)
else lb(m_g) $\leftarrow$ OR, nfr(m_g) $\leftarrow$ nd(m_g) $\cup$ nd(m),
nd(m_g) $\leftarrow$ nfr(m_g)
% see Fig.3.6.a %

4.b. Else % functor node m can be safely extended. See Fig.4.6.b and Fig. 4.6.c %

4.b.1. if nd(m) = {n}
then for i ∈ [1, k]
do create a new unexpanded leaf m/i as i^{th} son of m,
lb(m/i) ← lb(n/i)
od
if lb(n/i) = OR
then nfr(m/i) ← {(n/i)/1, ... , (n/i)/p}
else nfr(m/i) ← ∅
nd(m/i) ← {n/i} ∪ nfr(m/i)

4.b.2. else % assume nd(m) = $\{n_1, ..., n_p\}$ %
for i ∈ [1, k]
do create a new unexpanded leaf m/i for the i^{th} son of m,
if (∃ j ∈ [1, p]: lb(n_j/i) = max)
then lb(m/i) ← max, nd(m/i) ← $\{n_1/i, ..., n_p/i\}$
else lb(m/i) ← OR, nfr(m/i) ← $\{n_1/i, ..., n_p/i\}$,
nd(m/i) ← nfr(m/i)
od
S^{ul} ← (S^{ul} \ {m}) ∪ {m/1, ... , m/k}, S^{sn} ← S^{sn} ∪ {m}

Until S^{ul} = ∅
T' = compact(T')

With n_0 the root of type graph T, we use restrict(T) as a synonym for restrict(n_0). Note that a violation of the depth restriction can cause the creation of a backward arc. Consequently, the Algorithm 3.2. can introduce recursive types. The principal functor restriction enforces that the toplevel functor (principal label) of a given type is sufficient to determine to which son of an OR-node it belongs.

The following figures visualize the non trivial steps of Algorithm 3.2. Note that we still use the convention that node names starting with a n(m) refer to nodes in T(T'). If the labels of the nodes are not relevant only the node names appear in the figures.

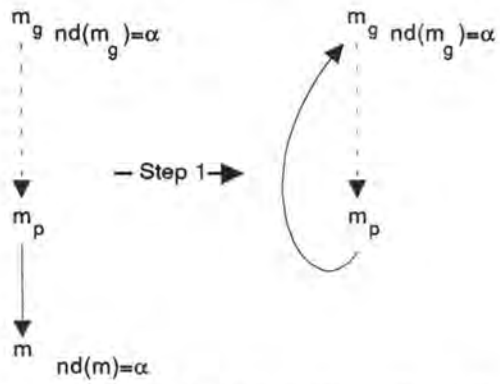


***Figure 3.4** : Step 1 of Algorithm 3.2.*

In Fig.3.4 the forward arc to the unexpanded leaf m is changed into a backward arc to m_g if it does not create a cycle consisting of only OR-nodes.

At the left-hand side in Fig.3.5 we represent a part of the original type graph, namely node n and some of its descendants. When the unexpanded leaf m is selected, nd(m) and nfr(m) are known. As involved-overlap(n, n_3) succeds, nd(m) and nfr(m) become respectively {n, n_1, n_2, n_3, n_{31}, n_{32}} and {n_1, n_2, n_{31}, n_{32}}. Step 3.b adds sons to m for the labels a/0, b/0 and f/1 if there does not exist an ancestor m_g of m such that safe-anc(m, nd, m_g) is true.

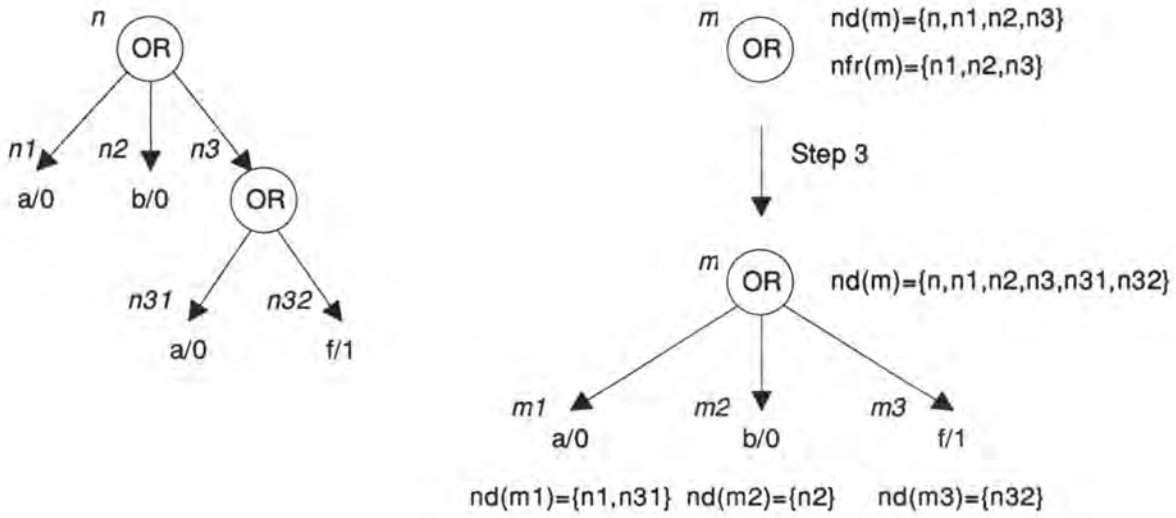
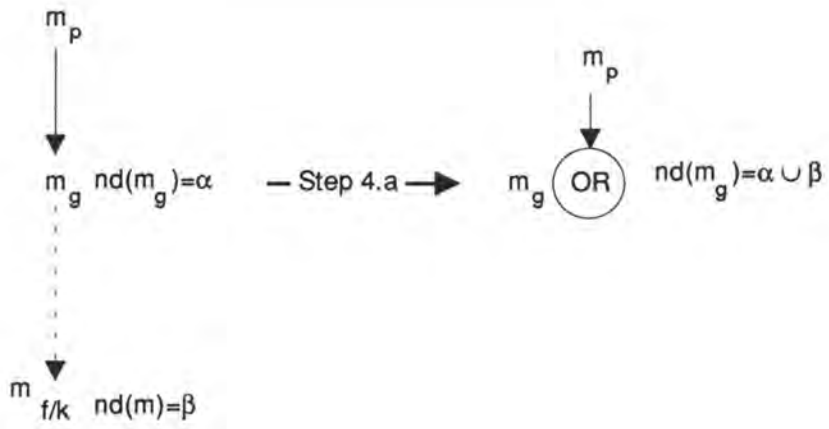


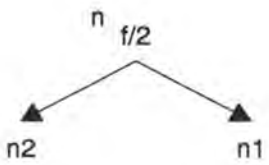
***Figure 3.5** : Step 3 of Algorithm 3.2.*

Suppose that the functor node m in Fig.3.6.a violates the depth restriction for f/k. Step 3.a.2 selects a ancestor m_g that has f/k as a principal label and the path from m_g to the parent of m does not consist of only OR-nodes. All the forward descendants of m_g and their arcs are removed. Finally m_g becomes an OR-node if it was not already one.
Fig.3.6.b and Fig.3.6.c illustrate step 4.b.1 respectively 4.b.2. The functor node m becomes a safe node as it can be extended without violation of the depth restriction for the functor f/k. Whether nd(m) contains one or more nodes, gives rise to different situations.

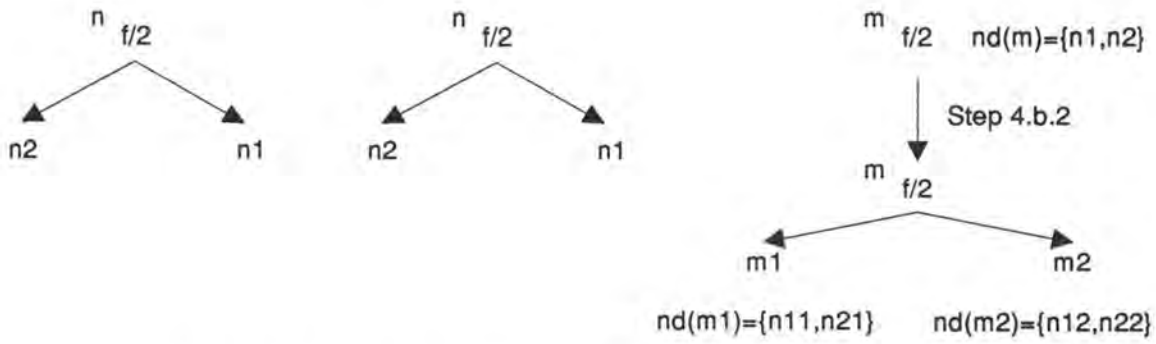
The computation of the restricted type graph corresponding to T_9 of Fig.3.3 is illustrated by Fig.3.7. The type graph is initialized with the node m_0. Step 3 adds m_1 and m_2 and step 2 makes a safe node of m_1. The node m_2 does not violate the principal functor restriction, so step 4.b.2 adds m_3 and m_4. Then step 3 adds two nodes m_5 and m_6. Now the depth restriction is violated by m_6, so we apply step 4.a.2 with m_2 as ancestor m_g. At this point $S^{sn} = \{m_0, m_1\}$ and $S^{ul} = \{m_2\}$, so m_2 is selected. The rest of the computation is straightforward. Note that when dealing with node m_9 step 3 adds a node m_{12} with $lb(m_{12}) = ./2$ and $nd(m_{12}) = \{n_7\}$. Now, the violation of the depth restriction is resolved by step 4.a.1 and a backward arc (m_9, m_7) is added as $nd(m_{12}) \subseteq nd(m_7)$.



3.6.a. Step 4.a.2: m is added as an alternative to one of its predecessors.

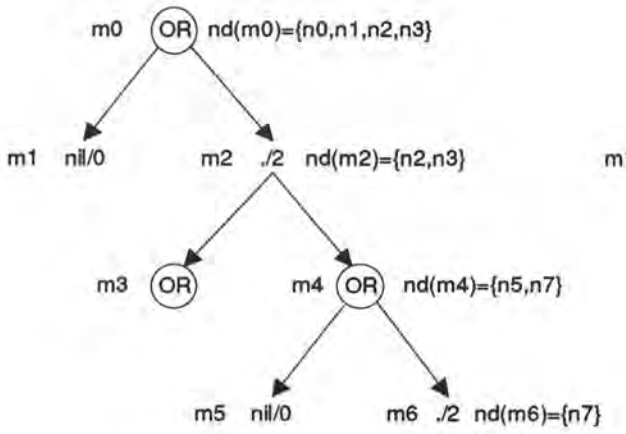


3.6.b. Step 4.b.1: nd(m) contains one functor node which can safely be added.

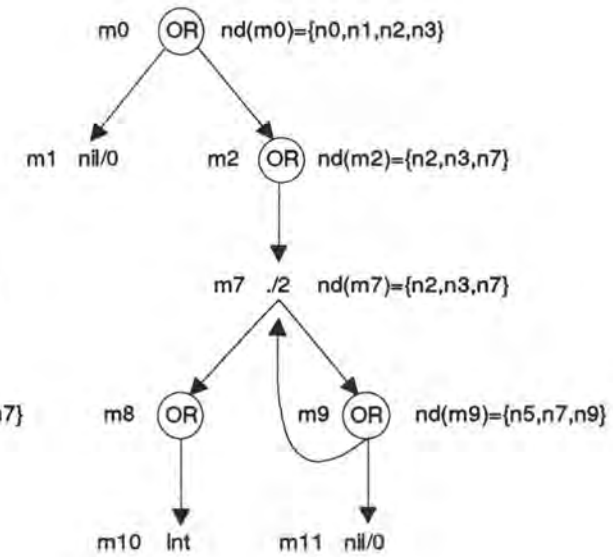


3.6.c. Step 4.b.2: nd(m) contains two functor nodes which can safely be added.

***Figure 3.6** : Step 4 of Algorithm 3.2.*



4.7.a. m6 violates the depth restriction



4.7.b. restrict(T9).

***Figure 3.7** : Algorithm 3.2: restrict(T_9)*

Next, we prove the correctness and the termination of algorithm 3.2. With respect to the correctness, we will show that ID(T') $\supseteq$ ID(n), but first we prove the following proposition.

Proposition 3.5.:

$\forall\, m \in S^{sn}$:

if lb(m) = OR then $\forall\, t \in$ ID-*nd(m)* $\exists\, i : t \in$ ID-*nd(m/i)* (1)

else if (lb(m) = f/k and k > 0)

then $\forall\, f(t_1, \ldots, t_k) \in$ ID-*nd(m)* : $(\forall\, i \in [1, k] : t_i \in$ ID-*nd(m/i))* (2)

else ID(*m*) = ID-*nd(m)* *% simple nodes and 0-ary functor nodes %*

(3)

Proposition 3.6.: *The Algorithm 3.2. constructs the restricted type graph T' such that* ID(*T'*) $\supseteq$ ID-*nd*(m_0) $\supseteq$ ID(*n*).

Proposition 3.7.: *The Algorithm restrict terminates.*

Definition 3.12.
A partially-ordered set (F, <) is said to be *well-founded* if there is no infinite increasing sequence of elements $f_1 < f_2 < \ldots$ from the set F.

Theorem 3.1.: *The Algorithm 3.2 terminates if there exists a well-founded set (F, <) and a function* $\tau : S_\tau \rightarrow F$ *(called the termination function) such that if* T'_{i+1} *is the intermediate state that follows immediately on* T'_i *then* $\tau\ (T'_i) < \tau\ (T'_{i+1})$.

3.2. Operations on normal type graphs

The normal type graphs will be used as a component of the abstract domains defined in Chapter 4. The framework imposes an algebraic structure on the abstract domain. In this section, some of those requirements are shown to be satisfied by the normal type graphs. From now on, we assume the type graphs to be normal.

3.2.1. Containments

We want to compare the denotation of two nodes in the same or two different type graphs. The call $\leq (n, m, \varnothing)$ of Algorithm 3.3. compares ID(n) with ID(m) and returns true iff $ID(n) \subseteq ID(m)$.

Algorithm 3.3 : $\leq (n, m, S^c)$

1. if $(n, m) \in S^c$ then true
2. else if lb(m) = max then true
3. else if (lb(n) = lb(m) = f/k and k > 0)
   then $\forall i \in [1, k]: \leq (n/i, m/i, S^c \cup \{(n, m)\})$
4. else if (lb(n) = lb(m) = OR with k = outdegree(n))
   then $\forall i \in [1, k]: \leq (n/i, m, S^c \cup \{(n, m)\})$
5. else if (lb(m) = OR and $\exists m_d \in pmd(m) : lb(m_d) = lb(n)$)
   then $\leq (n, m_d, S^c \cup \{(n, m)\})$
6. else lb(n) = lb(m)

An example is found in Fig.3.8.

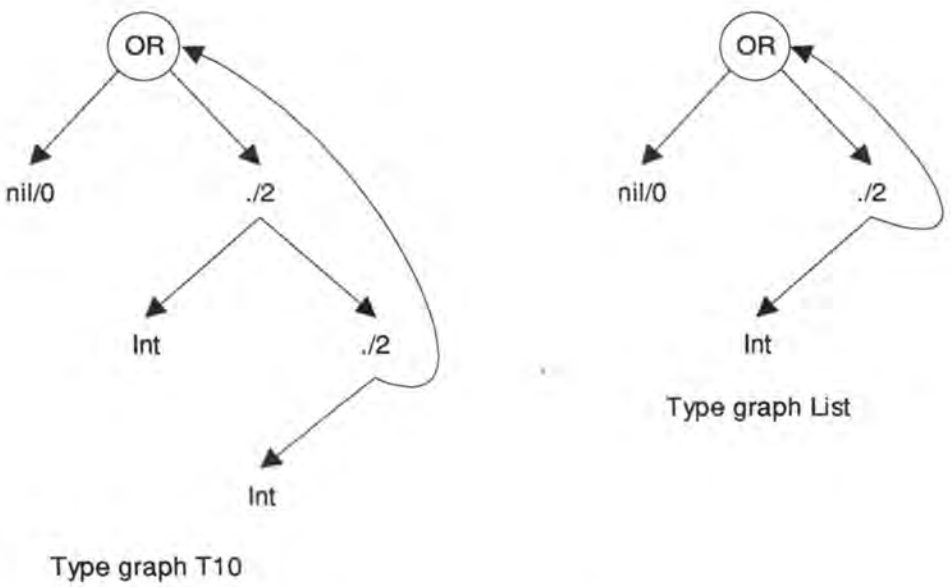


***Figure 3.8** : Algorithm 3.3: $T_{10} \leq$ List succeeds*

Proposition 3.9: *If a call* $\leq (n_i, m_j, \varnothing)$ *returns true and initiated a call* $\leq (n_p, m_q, S_1)$ *which returns true due to step 1 of the algorithm, then it also initiated a call* $\leq (n_p, m_q, S_0)$ *with* $S_0 \subseteq (S_1 \setminus \{ (n_p, m_q)\})$.

Proposition 3.10: $\leq (n, m, \varnothing)$ *iff* $ID(n) \subseteq ID(m)$ *(Correctness of* $\leq$*).*

Proposition 3.11: $T_m := max$ *is the maximal type graph.*

3.2.2. Equivalence

The equality of the denotations of the nodes n and m is computed by the call ≡ (n, m, ∅) which returns true iff ID(n) = ID(m).

Algorithm 3.4: ≡ (n, m, S^c)

1. if (n, m) ∈ S^c then true
2. else if (lb(n) = lb(m) = f/k and k > 0)
   then ∀ i ∈ [1, k] : ≡ (n/i, m/i, S^c ∪ {(n, m)})
3. else if (lb(n) = lb(m) = OR)
   then if prnd(n) = prnd(m)
   then ∀ n_i ∈ prnd(n) ∀ m_j ∈ prnd(m) : if $lb(n_i) = lb(m_j)$
   then ≡ (n_i, m_j, S^c ∪ {(n, m)})
   else false
4. else lb(n) = lb(nm)

Figure 3.9 gives an example.

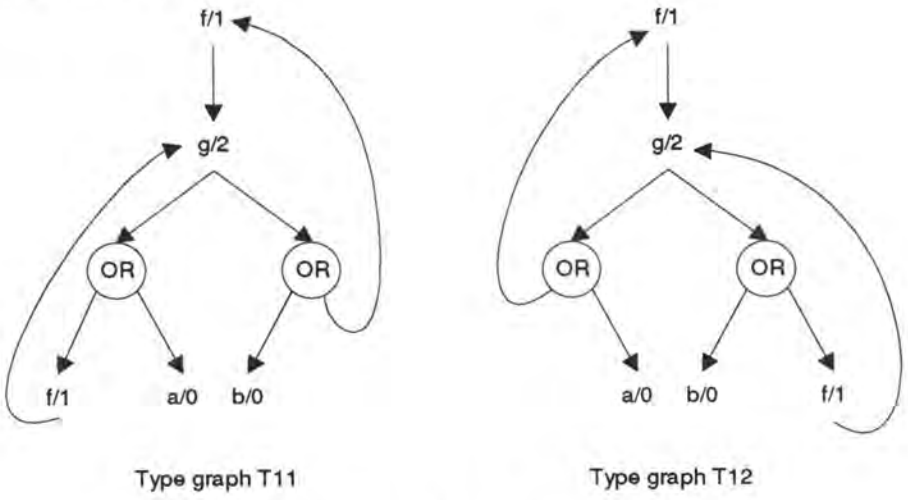


Figure 3.9 : *Algorithm 3.4.:* $T_{11} \equiv T_{12}$ *succeds*

Proposition 3.12: *If a call* $\equiv (n_i, m_j, \varnothing)$ *returns true and intitiated a call* $\equiv (n_p, m_q, S_1)$ *which returns true due to step 1 of the algorithm, then it also initiated a call* $\equiv (n_p, m_q, S_0)$ *with* $S_0 \subseteq (S_1 \setminus \{ (n_p, m_q)\})$.

Propostion 3.13: *(Correctness of* ≡*)* ≡ $(n, m, \varnothing)$ *iff* $ID(n) = ID(m)$.

3.2.3. Intersection

This operation computes a type graph T_{12} which is the *intersection* of the type graphs with roots n_1 and n_2, in the sense that its denotation contains only those terms which belong to $ID(n_1)$ and to $ID(n_2)$. Note that n_1 and n_2 may actually belong to the same type graph. The structure of the Algorithm 3.5 is similar to Algorithm 3.2. The given graph(s) is(are) left unchanged and the nodes of the type graph T_{12} under construction are either safe nodes ($\in S^{sn}$) or unexpanded leaves ($\in S^{ul}$). A node 1 of T_{12} has an associated function *is*. Its value, is(l), is a subset of $DESC^*(n_1) \cup DESC^*(n_2)$. In fact, *is(l)* always contains one or two nodes.

Definition 3.14.

$$ID\text{-}is(1) = \bigcap_{n \in is(1)} ID(n)$$

Proposition 3.14:
ID-*is(l)* =

if ($\forall\ n \in is(l)$: $lb(n) = max$) *then* S_{max}

else $\bigcap_{n \in ris(l)}$ ID(n) *with* $ris(l) = \{n \mid n \in is(l)$ *and* $lb(n) \neq max\}$

Algorithm 3.5: T_{12} = intersection(T_1, T_2)

Let n_0 and m_0 be the roots of respectively T_1 and T_2.

Initialization
T_{12} is initialized with a root node l_0.
$is(l_0) \leftarrow \{n_0, m_0\}$, $S^{sn} \leftarrow \varnothing$, $S^{ul} \leftarrow \{l_0\}$

repeat
Select a l from S^{ul} and let *ris(l)* be the set $\{n \mid n \in is(l)$ and $lb(n) \neq max\}$

1. if $\exists\ l_g \in S^{sn}$: safe-anc(l, is, l_g) then ul-barc(l, l_g)
2. else if $ris(l) = \varnothing$ % is(l) contains only max-nodes %
   then $lb(l) \leftarrow max$, $S^{ul} \leftarrow S^{ul} \setminus \{l\}$, $S^{sn} \leftarrow S^{sn} \cup \{l\}$
3. else if $\forall\ n \in ris(l) : lb(n) = OR$
   then $lb(l) \leftarrow OR$, $S^{ul} \leftarrow S^{ul} \setminus \{l\}$, $S^{sn} \leftarrow S^{sn} \cup \{l\}$,
   % suppose $m \in ris(l)$ %
   for $i \in [1, k]$ with $k = outdegree(m)$
   do create an unexpanded leaf l/i for the i^{th} son of l,
   $is(l/i) \leftarrow \{m/i\} \cup (ris(l) \setminus \{m\})$
   od
   $S^{ul} \leftarrow S^{ul} \cup \{l/1, \ldots, l/k\}$
4. else % $\exists\ n \in ris(l) : lb(n) = f/k$ or $lb(n) =$ Int, Real, ... %
   if $\forall\ m \in ris(l) \setminus \{n\} : (\exists\ m_d \in prnd(m) : lb(m_d) = lb(n))$
   then $lb(l) \leftarrow lb(n)$, $S^{ul} \leftarrow S^{ul} \setminus \{l\}$, $S^{sn} \leftarrow S^{sn} \cup \{l\}$,

if (lb(n) = f/k and k > 0)
then $S_{md} \leftarrow \{m_d \mid m \in ris(l) \setminus \{n\}$ and
$m_d \in pmd(m)$ and $lb(m_d) = lb(n)\}$
for i ∈ [1, k]
do create an unexpanded leaf l/i for the i^{th} son of l,
$is(l/i) \leftarrow \{n/i\} \cup \{m/i \mid m \in S_{md}\}$
od
$S^{ul} \leftarrow S^{ul} \cup \{l/1, \ldots, l/k\}$
else $lb(l) \leftarrow \perp$, $S^{ul} \leftarrow S^{ul} \setminus \{l\}$, $S^{sn} \leftarrow S^{sn} \cup \{l\}$

until $S^{ul} = \varnothing$
T_{12} = compact (T_{12})
% T_{12} is normal as the principal functor restriction cannot be violated. %

Example

T_4 = intersection (T_3, T_4)
T_{10} = intersection (T_{10}, T_3)

Next, we give the correctness and the termination proof for Algorithm 3.5, which are very similar to the proofs for Algorithm 3.2.

Proposition 3.15:
$\forall m \in S^{sn}$:

if(lb(m) = OR) then $\forall t \in$ ID-*is(m)* $\exists i : t \in$ ID-*is(m/i)* (1)
else if (lb(m) = f/k and k > 0)
then $\forall f(t_1, \ldots, t_k) \in$ ID-*is(m)* : ($\forall i \in [1, k] : t_i \in$ ID-*is(m/i)*) (2)
else ID(*m*) = ID-*is(m)* % *simple nodes and 0-ary functor nodes* % (3)

Proposition 3.16:
If T_{12} = *intersection* (n_1, n_2) *then* ($t \in$ ID(T_{12}) *iff* $t \in$ ID(n_1) *and* $t \in$ ID(n_2)).

Proposition 3.17:
The Algorithm intersection terminates.

Lemma 3.4: *(F, <) is a well founded set.*

3.2.4. Backward unification

During abstract interpretation it is possible that the denotation of the abstract success-substitution β_r of a call contains elements that are not instances of elements in the denotation of the abstract call-substitution β_i. The cause of this can be the use of Algorithm 3.2 for the computation of restricted types or the occurrence of a recursive call whose success-substitution is an upper bound. Some of these superfluous elements can be eliminated from β_r by *bacward unification* between β_r and β_i.
The core of the operation is the backward unification between the normal type T_r of a variable in β_r and its normal type T_i in β_i: T_n = btunif (T_i, T_r) such that ID(T_n) = $\{t_r \in ID(T_r) \mid \exists t_i \in ID(T_i) \exists \sigma : t_i \sigma = t_r\}$.

Proposition 3.18: $\{t_r \in ID(T_r) \mid \exists\, t_i \in ID(T_i)\ \exists\, \sigma : t_i\,\sigma = t_r\} = \{t \in ID(T_r) \mid t \in ID(T_i)\}$

Note that the latter set in Proposition 3.18 is just the denotation computed by intersection (T_i, T_r).

Algorithm 3.6: T_n = btunif(T_i, T_r) = intersection (T_i, T_r)

Proposition 3.19: *Every determinate selector s in T_i either exists in btunif(T_i, T_r) or btunif(T_i, T_r) = $\perp$.*

Fig.3.10 shows an example of backward unification. The nodes in T_n are adorned with their is-values. Step 1 of Algorithm intersection only creates a backward arc if there exists a safe ancestor with the same is-value as the unexpanded leaf at hand. But nodes with different is-values might have the same denotation. In this example we see that this is true for the nodes with is-values {1, 4} and {6}, and with is-values {3, 4} and {4}.

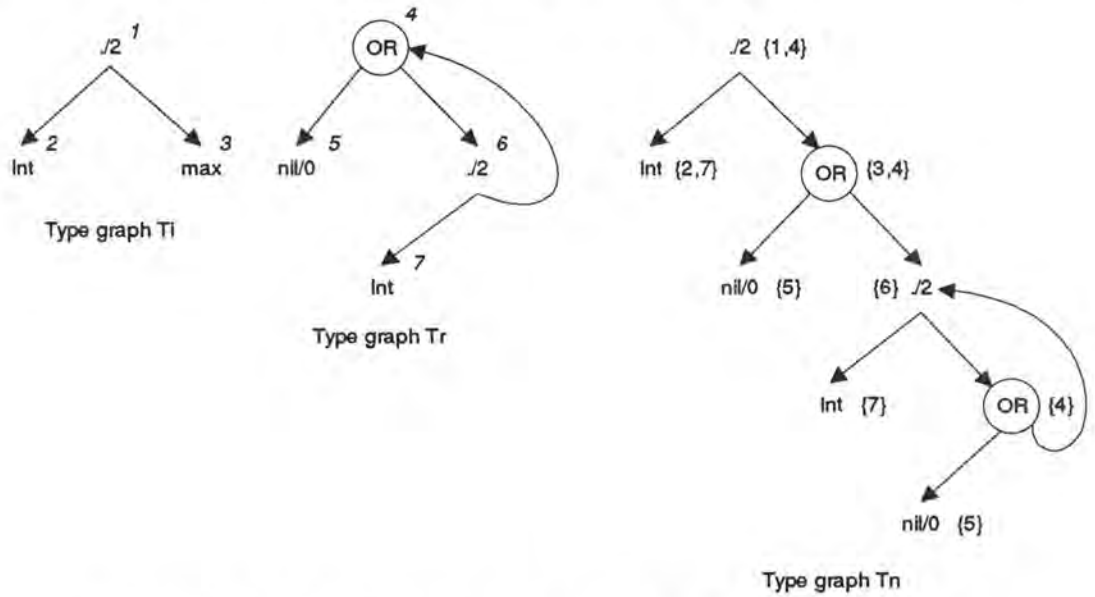


***Figure 3.10** : Example of backward unification: T_n = btunif(T_i, T_r)*

3.3 Expressive power of the type graphs

A rigid type is defined by a type graph describing a set of values. Due to the imposed restrictions (the principal label restriction and the depth restriction) it is not always possible to construct for a given set of values S, a type graph that has exactly S as its denotation: a type graph has to be used whose denotation is an overestimation of S.

The principal label restriction does not allow alternatives at an OR-node with the same principal label and roughly said, it forces those alternatives to be assembled in one son:
e.g. T = normalize(f(a, b) | f(c, d)) with T := f(a | c, b | d).

The loss of precision is acceptable as we still have information about the functor f/2 and its arguments. We do not longer have the information that if the first argument is a "a", then the second one is known to be a "c". This could be used during code generation to avoid unification at run time.

For the finiteness of the domain we have to impose the depth restriction on the type graphs. The choice of the depth constant for f/k is based on FDepth(f/k, t) with t the terms in the source text. This choice can be done for the whole program (taking into account the minimal value that differs from zero) or for each predicate separately which allows finer granularity. At the one hand, the expressiveness of the type graphs is limited: e.g. if the depth constant for functor ./2 is one, then Listone = restrict(OddList) with OddList := .(Int, nil | .(Int, OddList)) and ID(ListOne) $\supset$ ID(OddList). We encourage the reader to work out the computation of restrict(OddList).
At the other hand, the depth restriction introduces recursive types without any interaction with the user. The recursive types are indispensable for the detection of identical call-and success-substitutions.

3.4. Summary

In this chapter type graphs are introduced in order to describe sets of terms. From syntactic point of vue, type graphs are a special kind of graphs. They are rooted trees to which a special kind of arcs (backward arcs) is added in order to deal with recursive types. Furthermore, a label is associated with each node in the type graph. The structure of the type graph T together with the labels of its nodes determine the denotation of a node n in T, ID(n), which specifies the set of terms represented by n in T. The denotation defines the semantics of the type graphs. The rigid types, a synonym for the type graphs in this chapter, have the appropriate expressive power to be included in an abstract domain aimed at describing sets of values of variables. As the framework requires the abstract domain to have a certain algebraic structure and as the type graphs are intended to become a component of it, structural restrictions are imposed on the type graphs, thereby increasing the efficiency of operations on the type graphs, such as $T_1 \leq T_2$ and $T_1 \equiv T_2$. The operations are defined because they are needed to show that the abstract domain has the required properties.

Chapter 4

Abstract domain based upon rigid types

4.1. Definition of abstract substitutions

The aim of abstract interpretation is to compute for each program point an abstract substitution that is a correct approximation of the concrete substitution occurring at run time. A straightforward approach is to let an abstract substitution associate with each variable the set of values to which the variable possibly gets bound during the execution of the program. This is similar to the association of a single value to each variable in the concrete substitutions. The type graphs defined in the previous chapter are used to describe such sets of possible bindings. The main difference with the abstract domain based upon integrated types lays in the treatment of program variables that are still free or bound to partially instantiated values. In this version we approximate all the free values by the set of all possible terms (S_{max}). This approach justifies the term *rigid* introduced in Chapter 3.

The concrete substitutions also contains information about dependencies between values of program variables. If the abstract substitution would not include any information about such dependencies, the result of the abstract interpretation would still be valid but it would be a very imprecise approximation. Consider the case where X and Y are two variables whose possible bindings are given by the respective type graphs T_X and T_Y. After the unification of X and Y, we know that X and Y have the same value. The abstract interpretation computes a set of possible bindings for the values of X and Y after unification. This set is given by the "unifying" type graph T_u which is associated with X and Y. The fact that X and Y have the same value implies that the same term should be selected for X and Y from $ID(T_u)$ when considering the concrete substitutions approximated by the abstract substitution. Therefore we explicitly keep track in our abstract substitutions for which terms or subterms of the variables are *identical* in the concrete substitutions. This additional information substantially improves the precision of the abstract interpretation. The information is also useful for dealing with the explicit unifications in our normalized program (the arguments of the heads and the calls have to be variables). These explicit unifications create in fact the same kind of dependencies between values. In this case, the repeat previous call strategy [2] and [15] is an alternative solution to improve the precision.

Before formally defining abstract substitutions, we introduce some additional notational conventions.

In the concrete substitution $\{X \leftarrow t, Y \leftarrow Y, \ldots\}$, we call t the *value* of X and $Y \leftarrow Y$ represents the unbound variable Y. A substitution θ *over* some domain D is a substitution with $dom(\theta) = D$.

We introduce a notation to *select* nodes in a type graph.

Definition 4.1.
Let n be a node in the type graph.
$n/\in$ selects the node n itself.

$n/(f_1 / k_1, i_1). \cdots .(f_{p-1} / k_{p-1}, i_{p-1}).(f_p/k_p, i_p)$
with $1 \le i_1 \le k_1, \cdots, 1 \le i_p \le k_p$

selects the i_p^{th} son of the principal node with label f_p/k_p of the node

$n/(f_1/k_1, i_1). \cdots .(f_{p-1}/k_{p-1}, i_{p-1})$.

We call $\in$ and $(f_1/k_1, i_1). \cdots .(f_{p-1}/k_{p-1}, i_{p-1}).(f_p/k_p, i_p)$ *selectors*. If there are no OR-nodes on the path between n and the selected node, the selector $(f_1/k_1, i_1). \cdots .(f_{p-1}/k_{p-1}, i_{p-1}).(f_p/k_p, i_p)$ can be abbreviated by $i_1. \cdots .i_p$. The selector is said to be *determinate*. If there are OR-nodes on the path, the selector is said to be *non-determinate* and the full selector is required. To select a subterm in a specific term t the abbreviated selector is sufficient, e.g. with t the term f(g(a, b)), t/1.2 selects b. We only use selectors that are well defined for each step in the selection: there always exists an appropriate principal node or term to be selected. Selectors are denoted by the letters s and p possibly with a superscript or a subscript.

Definition 4.2.

$\forall s_1, s_2$ two selectors: $s_1 . s_2$ is the *concatenation* of the selectors s_1 and s_2

s_1 *extends* s_2 iff $\exists s_3 : s_1 = s_2.s_3$

s_1 and s_2 *overlap* iff s_1 extends s_2 or s_2 extends s_1

Example

According to Definition 4.2, the selector (f/2,2) . (g/3, 1) . (h/1,1) extends the selector (f/2,2) with s_3 = (g/3,1) . (h/1,1). Similarly, (f/2,1) extends $\in$ with s_3 = (f/2,1). The overlap property is a generalization of extend in the sense that it is no longer important which of the selectors extends the other: (f/2,1) and $\in$ overlap, and also (f/2,2) and (f/2,2) . (g/3,1) . (h/1,1) overlap.

We also introduce some notation concerning the substitutions. T_X denotes the type graph associated with X by an abstract substitution. We use the symbol n_0^x for the root of a type graph T_X. $ID(T_X)$ is a synonym of $ID(n_0^x)$. In a concrete substitution, terms are associated with variables belonging to its domain. X/s denotes the subterm t of the value of X such that $t \in ID(n_0^x/s)$. Note that in the latter case, s might need to be a non-determinate

selector. If confusion is possible between types of X at different program points, we use T_x^β and $n_0^{x/\beta}$ with β the abstract substitution at the program point. Similarly $X\theta/s$ explicitly refers to the subterm of the value of X in the concrete substitution θ.

Definition 4.3.
An abstract substitution β over a domain $D = \{X, Y, ...\}$ is either $\perp$ or a pair (type, sval).

The *TYPE-component*, type, associates a normal rigid type graph with each variables in the domain D and is selected by TYPE(β).
$TYPE(\beta) = \{X \leftarrow T_X \mid X \in D \text{ and } T_X \text{ is a non-empty normal rigid type graph}\}$

The *SVAL-component*, sval, is a set of *SVAL-constraints* and is selected by SVAL(β). SVAL is an acronym of Same VALue. A SVAL-constraint has the form $\{X/s_X, Y/s_Y\}$ with s_X and s_Y determinate selectors in respectively T_x^β and T_y^β and it expresses the fact that $X\theta/s_X = Y\theta/s_Y$ *must* hold in any concrete substitution θ represented by β. $\{X/s_1, X/s_2\}$ is possible but then $s_1 \neq s_2$.

By convention an abstract substitution β is denoted by $\perp$, if at least one of the types T_x^β is a $\perp$-node.

Definition 4.4.
A_D is the set of all abstract substitutions over domain D.

Definition 4.5.
$\{X/s_X, Y/s_Y\}$ is *less restrictive* than $\{X/p_X, Y/p_Y\}$ iff
$\exists\, s : s_X = p_X.s$ and $s_Y = p_Y.s$ and $s \neq \epsilon$

4.2. Denotation of an abstract substitution

The denotation of the abstract substitution β is defined by the concretization function γ which maps β into a set Θ of concrete substitutions in which each variable is bound to a value belonging to the denotation of its associated type graph and in which the dependencies between the values specified by the SVAL-component are satisfied.

Definition 4.6.
$\forall\, \beta \in A_D : \gamma(\beta) =$ if $\beta = \perp$ then $\varnothing$
else $\{\{X \leftarrow t_X\} \mid X \in D\} \mid$
$t_X \in ID(T_x^\beta)$ and
if $\{X/s_X, Y/s_Y\} \in SVAL(\beta)$ then $X\theta/s_X = Y\theta/s_Y\}$

4.3. Normal abstract substitutions

There are many SVAL-components that yield the same $\gamma(\beta)$. One of the reasons is that equality is a transitive relation, so, if for example $\{X/s_X, Y/s_Y\}$ and $\{Y/s_Y, Z/s_Z\}$ belong to SVAL(β), then $X\theta/s_X = Z\theta/s_Z$ with $\theta \in \gamma(\beta)$. Adding $\{X/s_X, Z/s_Z\}$ to SVAL(β) will not modify $\gamma(\beta)$. Also, X/s_X, Y/s_Y implies that $X/s_X \,.\, s = Y/s_Y \,.s$. It is desirable to have the SVAL-component in a form where constraints between any pair of variables are at the same time explicit and minimal, and to have the TYPE component in a form that is *compatible* with the SVAL-component, in the sense that

if $\{X/s_X, Y/s_Y\} \in$ SVAL(β) then $n_0^{x\beta}/s_X \equiv n_0^{y\beta}/s_Y$.

The Algorithm 4.2 establishes this, but first we give the Algorithm *replace(T, s_n, T')* that leaves the terms t in the denotation of the type graph T unchanged, except for the subterms t/s_n which are replaced by a term from ID(T'). Let n_0 be the root of T and let s_n and s_m be non-overlapping selectors. The existence of backward arcs makes it possible that ID(n_0/s_m) depends on ID(n_0/s_n). As only the subterms t/s_n are allowed to be changed, precautions have to be taken to preserve ID(n_0/s_m).

Algorithm 4.1: T_{new} = replace(T, s_n, T') with s_n a determinate selector

Let n_0 be the root of the type graph T, n be the node n_0/s_n and m_0 be the root of T'.
The type graphs T and T' are assumed to be normal.

if $s_n = \epsilon$ then $T_{new} = T'$
else

1. for each $p \in ANC^+(n)$
   do if $\exists$ backward arc (k, p) and $k \notin FDESC^+(n)$
   then T'' = construct(p) and the backward arc (k, p) is changed into (k, n_T) with n_T the root of T''
   od
2. $\forall\, p \in FDESC^*(n)$
   do remove node p and all the arcs (k, p) and (p, l)
   od
3. (n_p, n) is replaced by (n_p, m_0)
4. T_{new} = compact(T)

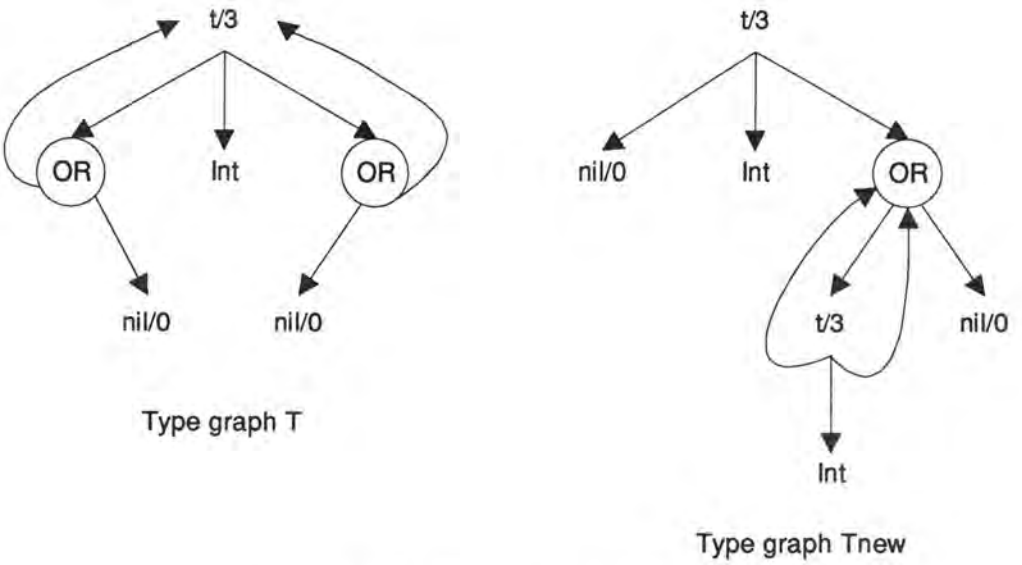


Figure 4.1 : T_{new} *= replace(T, 1, nil)*

Example
T_{new} = replace(T, 1, nil) with the type graphs given by Fig.4.1. Note that the depth restriction might be violated by T_{new}.

Proposition 4.1: *The Algorithm replace(T,* s_n*, T') constructs a normal type graph* T_{new} *such that for each term t* $\in$ ID*(T), there exists a term t'* $\in$ ID*(*T_{new}*) that is identical to t, except for the subterm* t/s_n *which is replaced by a term from* ID*(T').*

Algorithm 4.2: β_n = normalize(β)

Initially β_n equals β.
Perform one of the following transformation steps on β_n until no step is applicable:

R1 if $(\{X/s_X, Y/s_Y\}, \{X/s_X \,.\, s, Z/s_Z\} \in SVAL(\beta_n)$ and
$n_0^y/s_X \equiv n_0^y/s_Y$ and $n_0^x/s_X \,.\, s \equiv n_0^z/s_Z$ and
not $(\exists\{Y/p_Y, Z/p_Z\} \in SVAL(\beta_n)\ \exists\, p : p_Y \,.\, p = s_Y \,.\, s, p_Z \,.\, p = s_Z))$
then $\{Y/s_Y \,.\, s, Z/s_Z\}$ is added to $SVAL(\beta_n)$

R2 if $(lb(n_0^x/s_X) = lb(n_0^y/s_Y) = f/k$ and
$\forall\, i \in [1, k] : \{X/s_X \,.\, i, Y/s_Y \,.\, i\} \in SVAL(\beta_n)$ and $n_0^x/s_X \,.\, i \equiv n_0^y/s_Y \,.\, i)$
then $\{X/s_X, Y/s_Y\}$ is added to $SVAL(\beta_n)$

R3 if $(\{X/s_X, Y/s_Y\}, \{X/s_X \,.\, s, Y/s_Y \,.\, s\} \in SVAL(\beta_n)$ and $s \neq \in)$
then the latter element is removed from $SVALB(\beta_n)$

R4 if $(\{X/s_X,\ X/s_X \,.\, s\} \in SVAL(\beta_n)$ and $s \neq \in)$ then β_n becomes $\perp$

R5 if $(\{X/s_X, Y/s_Y\} \in SVAL(\beta_n)$ and

(if X = Y then s_X and s_Y do not overlap) and not ($n_0^x / s_X \equiv n_0^y / s_Y$))

then T = intersection(n_0^x / s_X , n_0^y / s_Y) and

$T_x^{\beta n}$ = replace($T_x^{\beta n}$, s_X, T) and $T_y^{\beta n}$ = replace($T_y^{\beta n}$, s_Y, T)

and if (lb($T_x^{\beta n}$) = ⊥ or lb($T_y^{\beta n}$) = ⊥) then β_n becomes ⊥

Observe that R5 establishes the compatibility between the SVAL- and the TYPE-component, whereas rules R1 and R2 only make implicit SVAL-constraints explicit if the existing SVAL-constraints are compatible with the TYPE-component.

Example

TYPE(β) = {X ← T_X, Y ← T_Y} with T_X := f(alb,alc) and Ty := g(alb, Int)
SVAL(β) = {{X/1, Y/1}, {X/1, X/2}}
Initially, β_n is equal to β.

First {X/1, X/2} causes the application of R5 as $n_0^x /1 \equiv n_0^x /2$ does not hold in β_n. The result of intersection($n_0^x /1$, $n_0^x /2$) is the type graph T:=a. After the calls to replace, $T_x^{\beta n}$:= f(a, a).

Now also {X/1, Y/1} activates R5 as $n_0^x /1 \equiv n_0^y /1$ does not hold in β_n. The intersection is the type graph T:= a and $T_y^{\beta n}$ becomes g(a, Int).

Finally R1 detects that also its type conditions are fulfilled and {X/2, Y/1} is added to SVAL(β_n). Thus, β_n = normalize(β) is given by:
TYPE(β_n) = {X ← T'_X, Y ← T'_Y} with T'_X := f(a, a) and T'_Y := g(a, Int)
SVAL(βn) = {{X/1, Y/1}, {X/1, X/2}, {X/2, Y/1}}

The implementation of Algorithm normalize is organized as follows. We first check whether no explicit circularities exist, if so β_n becomes ⊥. Then, we force compatibility between the types and the SVAL-constraints in β by R5. From now on, we only have to take into account R1 to R4 without the conditions imposed on the types, because the compatibility exists not only for the explicit SVAL-constraints (in SVAL(β)), but also for the implicit ones (still to be added), due to the transitivity of the equality relation. We work with two sets of SVAL-constraints: svnew, initially equal to SVAL(β), and safe, initially the empty set. We move constraints from svnew to safe while we preserve for safe the invariant that R3 nor R4 are applicable on constraints in safe and that the application or R1 or R2 on constraints in safe does not change safe ∪ svnew. Note that removing a constraint from safe does not violate the invariant. So, eventually, β_n becomes ⊥, or svnew becomes empty and safe is the set of of SVAL-constraints in its explicit and minimal form compatible with the types.

Applying rules R1, R2, R3, R4 and R5 yields the *normal* form for the abstract substitution. However, it is not a unique form: type graphs do not have a unique form, moreover abstract substitutions that differ only in SVAL(β), can have the same γ(β). Consider for example a type T with #ID(T) = 1 and TYPE(β) = {X ← T, Y ← T}. The presence of {X/∈, Y/∈} in SVAL(β) does not affect γ(β). From now on we assume that abstract substitutions are in normal form.

We give some examples of abstract substitutions over the domain {X, Y}.

1. $TYPE(\beta_1) = \{X \leftarrow T_x, Y \leftarrow T_y\}$ with $T_x := f(alb, alc)$ and $Ty := g(alb, Int)$
   $SVAL(\beta_1) = \varnothing$
   Although the SVAL-component enforces no dependencies between the values of X and Y, X and Y may have identical subterms:
   $\theta_1 = \{X \leftarrow f(a, a), Y \leftarrow g(b, 3)\} \in \gamma(\beta_1)$
   $\theta_2 = \{X \leftarrow f(a, a), Y \leftarrow g(a, 3)\} \in \gamma(\beta_1)$
   $\theta_3 = \{X \leftarrow f(a, c), Y \leftarrow g(a, 3)\} \in \gamma(\beta_1)$

2. $TYPE(\beta_2) = TYPE(\beta_1)$ and $SVAL(\beta_2) = \{\{X/1, Y/1\}\}$
   The SVAL-component enforces the first arguments of the values of X and Y to be identical, thus $\gamma(\beta_2) \subseteq \gamma(\beta_1)$.
   θ_2 and $\theta_3 \in \gamma(\beta_2)$, but $\theta_1 \notin \gamma(\beta_2)$.

3. $TYPE(\beta 3) = TYPE(\beta_1)$ and $SVAL(\beta_3) = \{\{X/1, Y/1\}, \{X/1, X/2\}\}$
   β'_3 = normalize β_3 as is computed in the previous example.
   $TYPE(\beta'_3) = \{X \leftarrow T'_x, Y \leftarrow T'_y\}$ with $T'_x := f(a, a)$ and $T'_y := g(a, Int)$
   $SVAL(\beta'_3) = \{\{X/1, Y/1\}, \{X/1, X/2\}, \{X/2, Y/1\}\}$
   Now $\theta_2 \in \gamma(\beta_3) = \gamma(\beta'_3)$, but $\theta_3 \notin \gamma(\beta'_3)$.

We can avoid to introduce type names for non recursive types that occur only once such as T_x, T_y, T'_x and T'_y in the examples above by replacing the type names in the abstract substitution by the right-hand side of their context free grammar.
For example, $TYPE(\beta'_3) = \{X \leftarrow f(a, a), Y \leftarrow g(a, Int)\}$.

4.4. Operations on abstract substitutions in normal form.

In this section we define and verify the algebraic structure imposed by the framework, for the abstract domain of normal abstract substitutions.

Definition 4.7.
$A_N = \{\beta \mid \beta \in A_D$ and β is in normal form$\}$

4.4.1. Containment

We need an order relation $\leq$ between abstract substitutions over the same domain D with the following property : if $\beta \leq \delta$ then $\gamma(\beta) \subseteq \gamma(\delta)$.

Definition 4.8.
$\forall \beta, \delta \in A_N$:
$\beta \leq \delta$ iff
$\beta = \bot$ or
$\forall X \in D$:
$T_x^\beta \leq T_x^\delta$ and
$\forall \{X/s_X, Y/s_Y\} \in SVAL(\delta)\ \exists \{X/p_X, Y/p_Y\} \in SVAL(\beta)\ \exists s$:
$s_X = p_X . s$ and $sy = p_Y . s$

Proposition 4.3: $\forall \beta, \delta \in A_N: \beta \leq \delta \Rightarrow \gamma(\beta) \subseteq \gamma(\delta)$.

Proposition 4.4.: $\beta_{max} = (\{X \leftarrow max \mid X \in D\}, \varnothing)$ is the maximal element of A_N.

4.4.2. Equivalence

Definition 4.9.
$\forall \beta, \delta \in A_N: \beta \equiv \delta$ iff $\beta \leq \delta$ and $\delta \leq \beta$

Proposition 4.5: $\forall \beta, \delta \in A_N: \beta \equiv \delta \Rightarrow \gamma(\beta) = \gamma(\delta)$.

Notice that the equivalence classes defined by $\equiv$ are not singletons because the equivalence classes over the types are not $\leq$ is a partial order over the equivalence classes defined by $\equiv$.

4.4.3. The upper bound operation

Definition 4.10.
$\forall \beta_1, \beta_2 \in A_N$:
$upp(\beta_1, \beta_2) =$
if $\beta_1 = \bot$ then β_2
else if $\beta_2 = \bot$ then β_1
else δ with

$TYPE(\delta) =$
$\{X \leftarrow normalize(n_X) \mid X \in D$ and
n_X is an OR-node with two outgoing forward arcs:
one of the root of $T_x^{\beta 1}$ and one to the root of $T_x^{\beta 2}\}$

$SVAL(\delta) = \{\{X/s_X, Y/s_Y\} \mid$
$\beta_p, \beta_s \in \{\beta_1, \beta_2\}$ and $\beta_p \neq \beta_s$ and
$\exists \{X/s_X, Y/s_Y\} \in SVAL(\beta_s)\ \exists \{X/p_X, Y/p_Y\} \in SVAL(\beta_p)\ \exists s$:
$s_X = p_X . s$ and $s_Y = p_Y . s$ and
s_X respectively s_Y are determinate selectors in T_x^δ respectively $T_y^\delta\}$

$\gamma(\delta)$ includes all concrete substitutions of β_1 and β_2. $ID(T_x^\delta)$ is a superset of the union of $ID(T_x^{\beta 1})$ and $ID(T_x^{\beta 2})$. Consequently, the selectors appearing in $SVAL(\beta_1)$ ans $SVAL(\beta_2)$ are still defined in δ, although they might be non-determinate. The restrictions concerning identical (sub)terms are imposed by the SVAL-constraints and still hold in δ if they exist in β_1 and in β_2. If $\beta_1(\beta_2)$ contains a less restrictive constraint than $\beta_2(\beta_1)$, then the one in the former occurs in δ too. Furthermore, the SVAL-constraint must have determinate selectors in T_x^δ and in T_y^δ by Definition 4.3.

Proposition 4.6:
$\forall \beta_1, \beta_2 \in A_N$: $\exists$ upp$(\beta_1, \beta_2) \in A_N \Rightarrow$ upp$(\beta_1, \beta_2) \geq \beta_1$ and upp$(\beta_1, \beta_2) \geq \beta_2$.

Proposition 4.7:
$\forall \beta_1, \beta_2 \in A_N$: β_1, β_2 are in normal form, $\Rightarrow \delta =$ upp(β_1, β_2) is also in normal form.

4.4.4 A finite subdomain

We have an infinite number of normal types, so obviously, we also have an infinite number of different abstract substitutions over the same domain. To obtain a finite subdomain, we impose the depth restriction on the functors. R is the operation transforming an abstract substitution to a depth restricted abstract substitution. $\delta = R(\beta)$ is defined as follows:

Definition 4.11.
$\forall \beta \in A_N$: $R(\beta)$ = if $\beta = \perp$ then $\perp$
else δ with

$TYPE(\delta) = \{X \leftarrow \text{restrict}(T_x^\beta) \mid X \in D\}$
$SVAL(\delta) = \{\{X/s_X, Y/s_y\} \mid$
$\{X/s_X, Y/s_y\} \in SVAL(\beta)$ and
s_X and s_y are determinate selectors in respectively T_x^δ and T_y^δ
and $n_0^{x\delta}/s_X \equiv n_0^{y\delta}/s_y\}$

Note that SVAL-constraints are certainly removed when s_X or s_y violates the depth bound for a functor because the path is now circular in δ and contains at least one OR-node (Proposition 3.3). Note that R5 of Algorithm 4.2 cannot be used to force compatibility between the types and the SVAL-constraint in δ as is illustrated by the first example bellow.

Proposition 4.8: $\forall \beta \in A_N$: $\beta \leq R(\beta) \Rightarrow R(\beta)$ is in normal form.

Proposition 4.9: The number of $\equiv$ equivalence classes of depth restricted abstract substitutions over a finite domain D for a finite set of functors is finite.

Examples

1. TYPE(β) = {X ← .(T_0, . (T_1, nil)), Y ← T_0}
   SVAL(β) = {{X /(./2, 1), Y / ∈}}
   With T_0 and T_1 rigid types with non-overlapping sets of principal functors and with 1 as depth bound of ./2, the computation of R(β) first calls restrict(T_x^β) which results in T_3 := . (t_0 | T_1, nil | T_3). The SVAL-constraint is dropped because T_3/(./2, 1) ≡ T_0 / ∈ does not hold. Thus δ = R(β) is given by:
   TYPE(δ) = {X ← T_3, Y ← T_0} and SVAL(δ) = ∅
   If we would retain {X /(./2, 1), Y / ∈} in SVAL(δ), then R5 in δ_n = normalize(δ) could force compatibility between the SVAL-constraint and the types. However, this operation is not valid as $T_x^{\delta n}$:= .(T_0, nil | T_3). $T_x^{\delta n}$ again violates the depth restriction for ./2 and T_3 = restrict($T_x^{\delta n}$).
2. Let List and ListOne be the types defined in section 3.1.2 and consider the abstract substitution β, defined as follows:
   TYPE(β) = {X ← . (Int, nil | .(Int, nil)), Y ← List, Z ← LisOne}
   SVAL(β) = {{X/(./2, 1), Z/(./2, 1)}}
   δ = R(β) with 1 for the depth bound of ./2
   TYPE(δ) = {X ← ListOne, Y ← List, Z ← ListOne}
   SVAL(δ) = {{X/(./2, 1), Z/(./2, 1)}}

4.4.5 Backward unification

Backward unification between β_i and β_r enables us to eliminate some of the superfluous concrete substitutions from $\gamma(\beta_r)$ with β_i the abstract call-substitution and β_r the abstract success substitution.

Definition 4.12.
∀ β_i, β_r ∈ A_N:

bunif(β_i, β_r) = if (β_i = ⊥ or β_r = ⊥) then ⊥
else normalize(β_n) with

TYPE(β_n) = {X ← btunif($T_x^{\beta i}$, $T_x^{\beta r}$) | X ∈ D}
SVAL(β_n) = SVAL(β_r) ∪ SVAL(β_i)

Proposition 4.10:
If ∃ θ_r ∈ $\gamma(\beta_r)$ ∃ θ_i ∈ $\gamma(\beta_i)$ ∃ σ : θ_i σ = θ_r *then* θ_r ∈ γ(*bunif*(β_i, β_r)).

Examples

1. Given β_{in} and β_r :
   TYPE(β_{in}) = {X ← T_x^1 } with T_x^1 := f(max) and SVAL(β_{in}) = ∅
   TYPE(β_r) = {X ← T_x^2 } with T_x^2 := a | f(T_x^2) and SVAL(β_r) = ∅
   β_n = bunif(β_{in}, β_r) with
   TYPE β_n = {X ← T_x^3 } with T_x^3 := f(a | T_x^3) and SVAL(β_{in}) = ∅

2. Given β_{in} and β_r :
   $TYPE(\beta_{in}) = \{X \leftarrow List, Y \leftarrow T_y, Z \leftarrow max\}$ with T_y := .(Int, .(Int, nil))
   and $SVAL(\beta_{in}) = \varnothing$
   $TYPE(\beta_r) = \{X \leftarrow List, Y \leftarrow ListOne, Z \leftarrow ListOne\}$ and $SVAL(\beta_r) = \varnothing$
   $\beta_n = bunif(\beta_{in}, \beta_r)$ with
   $TYPE(\beta_n) = \{X \leftarrow List, Y \leftarrow T_y, Z \leftarrow ListOne\}$ and $SVAL(\beta_n) = \varnothing$

4.5 Summary

An abstract substitution β describes a set of concrete substitutions θ. The first component of the abstract substitutions defined in this chapter is called the TYPE-component and associates with each program variable a rigid type, whose denotation is a set of terms. Furthermore, dependencies between values of program variables in the concrete substitutions are encoded by the second component of the abstract substitutions: SVAL-constraints (SVAL is an acronym of SameVALue) specify which (sub)terms in the values of the variables must be identical. Consider, for instance, the unification 'Z = X.Y'. After a successful unification the values of Z all have toplevel functor ./2 whose first (second) argument has the same value as program variable X(Y). This kind of dependencies is expressed by the SVAL-component. The normal form of an abstract substitution forces the SVAL-constraints to be explicit and minimal and forces compatibility between types and SVAL-constraints. This form is useful when specifying the domain dependent operations for the abstract interpretation. The abstract domain consisting of normal abstract substitutions is shown to have the required algebraic structure.

Chapter 5

Implementation of the type-graphs

5.1. Introduction

In the first section of this chapter, we will discuss the data structures chosen for the representation of the type-graphs. They are two types of algorithms in the thesis[2], the first group is declarative oriented in its specification in the sense that it is a list of rules which describe the algorithm. The second group is more classical in its specification, they are written in a pseudo-mathematical notation which can reserve some surprise during the implementation.

The second section proposes a new version of the algorithm Compact. This part begins by the formalization of the procedure non_empty_ID and of all the tools which allow us to construct our version of Compact.

The section 5.3 contains the motivations to keep the algorithm Restrict as presented in the thesis, and it is followed (in section 5.4) by a description of the main principles used for the implementation of the other algorithms proposed.

5.2. Representation of the type-graphs

5.2.1. Discussion on a data structure for the type-graphs

Type-graphs as described in the thesis can be represented in several manners. We could use a representation by means of static arrays, trees... The first decision is to choose a representation which is able to meet all the possible configurations of the type-graphs. Because of the unpredictable character of the size of a type-graph, the idea to employ static memory representation was aborted. As matter of fact, a type-graphs can have n sons attached to its root and another can have a depth of m, as expressed by the next figure.

---

[2] By thesis, we always reference [7].

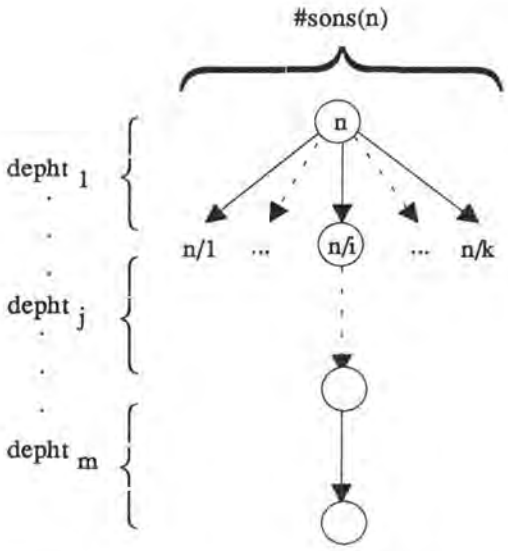


***Figure 5.1** : Size of a type-graph*

The most powerful representation was to use a n-tree (tree were each node can have n-sons) to represent the type-graphs. The tree's structure is ideal for this kind of representation because the tree can be extended almost until infinite, we can add that the n-tree was mandatory because it was not possible to determine the number of branchs the tree should have.

Once the type of the data structure is chosen, we had to construct the data structure itself. Observing the type-graphs and its definition presented in the previous chapter, we can point out the following particularities :

- there exists three groups of nodes. The first one is composed of the nodes which have got any son (constant-node or functor with arity null), the next one is the functor-node (or functor with arity different of null) and finally we have the OR-node (or functor with arity different of null, in which the order of the son has no importance).
- those nodes can generate two types of arcs. We can differ the backward arcs which are always pointing to a functor-node or to an OR-node from the forward arcs which can point to any kind of node.
- the simple-node group can be divided in two parts. The first one is composed by the fundamental types (Int, Real, Max, Bottom, [ ]) and the other one is composed by the other types (a/0,b/0 for example).

Regarding those considerations, we can observe that there are two principal kinds of component, those representing functors and those presenting arcs. We can point out that :

- it is not necessary to distinct functors with a zero arity (constants and fundamental types) from functors with a non-zero arity, the only difference between both is that the first kind of functor does not generate arcs. Furthermore, there is no reason to separate the OR-nodes from the other functor-node, the only change between both is that the order of the sons has importance for the functor-nodes.
- as for the functor-node it is not mandatory to create a specific data structure for each kind of arc. This is due to their resemblance, indeed, both point to a functor-node. An Arc is thus a specific kind of cell which contains a reference to a functor-node and a direction

(Backward or Forward). To differenciate the nodes denoting a functor from the cell denoting arcs, we will call the first one the functor-node ans the second one the sons.

So, the type-graphs could be represented by only 2 types of cells, those denoting functors in general, and those denoting the sons of the functor.

Before discussing the organisation of the trees, we will define a notation for the type-graph. Manipulating the type-graph by means of their graphical representation is certainly more easy to understand but will be too embarrassing. Furthermore, the implementation of a graphical representation was not the purpose of this work.

5.2.2. Formalization of the Context Free Grammar

As expressed in the chapter 4, the information described by a type-graph can be expressed by a context free grammar. The following B.N.F. notation formalizes the notation employed in the rest of this paper.

<Result> ::= {<Affect>}
<Affect> ::= <Name> := <Tree>
<Tree> ::= <Constant-Node> |
<Fundamental-Node> |
<Functor-Node> <$Name_{i1}$>...<$Name_{in}$>

Example :
Let X be a type-graph associated with the variable represented by the following figure.

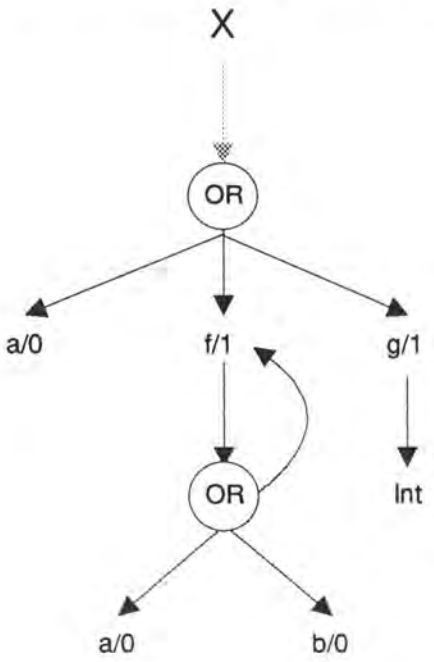


***Figure 5.2** : Example of type-graph*

This can be expressed in our notation by the next expression.

$$T_0 := OR(a, T_1, g(Int))$$
$$T_1 := f(OR(a, T_1, b))$$

It could be remarked that the notation proposed in the thesis has been a little changed. We prefer the prefixed notation to the infixed notation for the OR-node. They are two reasons for this, the first one is derived from the chosen data structure, our decision to consider the OR-node as a functor node induct the same representation, so there is no need to choose an other notation specially for the OR-node. The second reason is less academic, we found the prefixed notation slightly more readable than the infixed.

5.2.3. Definition of the data structure chosen for the type-graphs

All the functor-nodes are bearing a name, the size of the name varies from a functor to another. So, it could be more clever to store all the functors names into an array and to memorize the entry of the array as the name of the functor. An example can be found in the next figure[3].

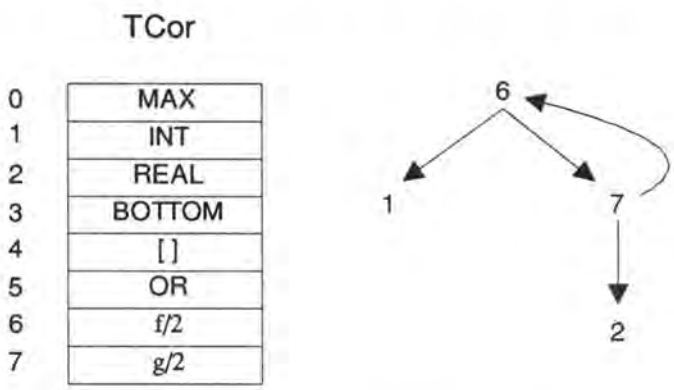


***Figure 5.3** : The array TCor*

This permits to store the name of the functor-node only once and it allows to replace the comparaison of two string by the comparaison of two integer which is more faster.

It follows a complete description of the data structure chosen to represent the type-graphs.

[3] Notation : [ ] denotes an empty list.

Cell representing a functor-node

Id_Name	Color	Son

where :

Id_Name is an integer used to represent the entry corresponding to the name of the functor in array *TCor*.

Color is an integer which will be used to sort the tree (see later).

Son is a pointer to the list of the sons attached to the functor-node (the pointer is set to Null if the functor-node has no sons).

Cell representing the sons of a functor-node

Type	Node	NextSon

where :

Type is a flag indicating the direction of the arc, so it may contain two different values : 'F' for the forward arc, and 'B' for the backward arc.

Node is a pointer to a functor-node.

NextSon is a pointer to the next son of the list.

Example :

Let X, be a variable to which is associated the type-graph $T_X := F(Int, G(Real, T_X))$.

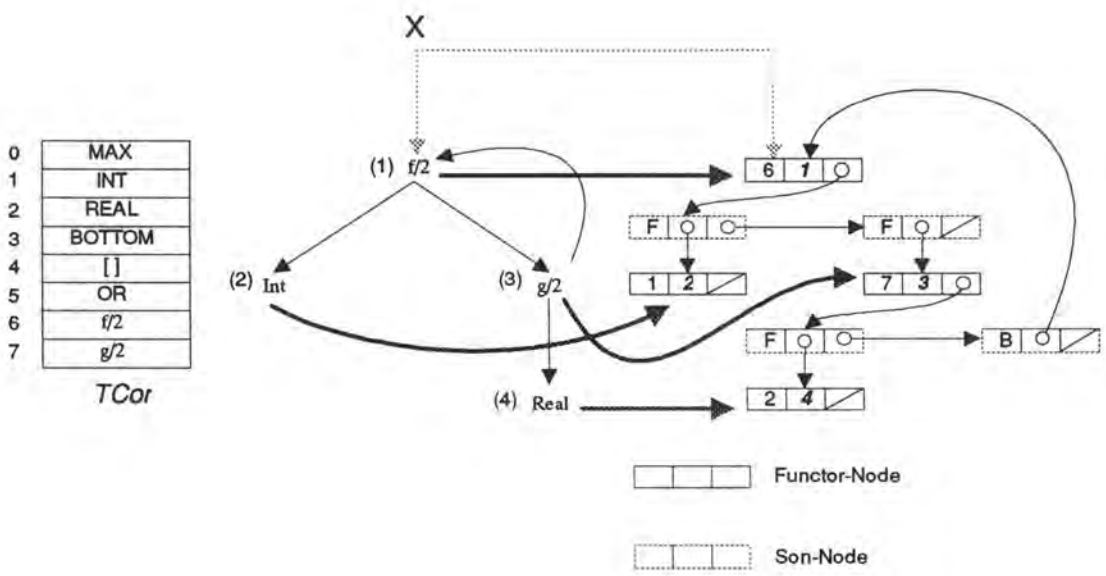


***Figure 5.4** : Example of the chosen data structure*

This figure requires some comments, as explained before, functor-nodes are composed of three fields, the first and the third one do not need more explanations. The second one permits the coloration of the tree, it is numbered starting form the root to each leaf by a depth first search procedure. The root is labelled to one, the other node by an incrementation by one. This method permits to have an approximation of the location of a node in the tree compared to another. It is only an approximation because this artifice can just permit to know if a node has been visited before another in a depth first search, as expressed by the following figure.

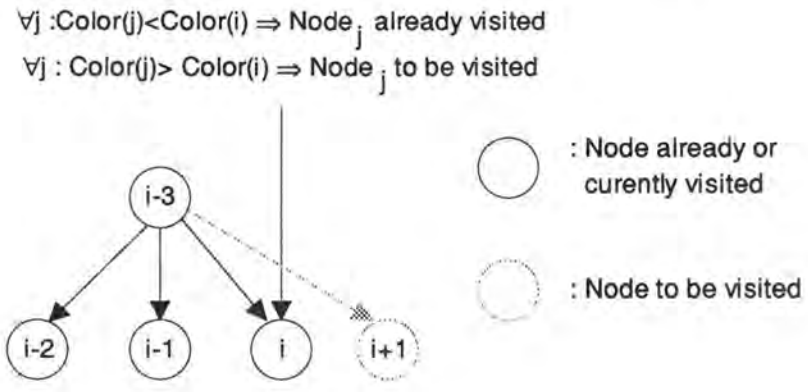


***Figure 5.5** : Utility of the field color*

5.3. *Compacted type-graphs*

The aim of this algorithm (Algorithm 3.1) is to modify the tree so that it satisfies the eight rules presented in the thesis. Starting form a tree T_{source} we have to build the compacted tree $T_{dest.}$. The algorithm can be inefficient for an implementation in C if translated as proposed because of the loop "Perform one of the following transformation steps on T_{source} until no step is applicable" .

We meet here the main problem of the adaptation of some of the proposed algorithms into an imperative language. The algorithms were conceived to be implemented in Prolog which is a declarative language. However, it has to be possible to translate an algorithm conceived for Prolog into an Imperative Language but it will be as hard as inefficient.

Our algorithm is constructed on two passes. The first pass deals with the rules 1 to 5 and with the rule 7. This pass treats all the fundamental nodes and all the OR-nodes which aren't in a path of OR-node. The second pass deals with those chains and with the backward arcs. Before a detailed description of the algorithm, we will focus the implementation of the function non_empty_ID. The resulting function will facilitate the construction of Compact.

5.3.1. Implementation of non_empty_ID

In [7], as an introduction to the algorithm Compact, the author presents a boolean function called non_empty_ID. Non_empty_ID(n) is true if ID(n) is a non-empty set of finite terms. She proposes the following recursive definition.

$$
\begin{aligned}
&\text{non\_empty\_ID}(n) \text{ iff}\\
&\text{if } (\text{lb}(n) = f/k \text{ and } k > 0)\\
&\quad \text{then } \forall i \in [1,k] : \text{non\_empty\_ID}(n/i)\\
&\quad \text{else if } \text{lb}(n) = \text{OR}\\
&\qquad \text{then } \exists i : \text{non\_empty\_ID}(n/i)\\
&\qquad \text{else } \text{lb}(n) \neq \bot
\end{aligned}
$$

The problem, as implicitly suggested in the thesis, is that the implementation of this definition can create easily an infinite loop as expressed by the following figure. Suppose that we apply the definition in the order of the sons of the root, doing like this generates an infinite loop situation because we always choose the backward arc.

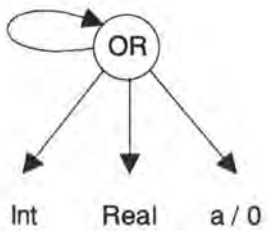


***Figure 5.6** : T0 := OR(T0,Int,Real,a/0)*

To avoid this situation, the author proposed a bottom-up approach which permits to solve the previous problem. But this approach can not always be performed as it can be shown in the next figure.

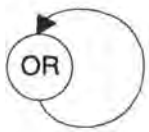


***Figure 5.7** : T := OR(...OR()...)*

Thus, we have to construct a radically different algorithm than the one suggested by the previous definition. We will, by means of the example beneath, draw some establishment which will later be used in our implementation. It represents the type-graph associated with the variable X. Suppose that we have to determine if its denotation is a non-empty set of finite terms. To do this, we have to know the denotation of each son of the root.

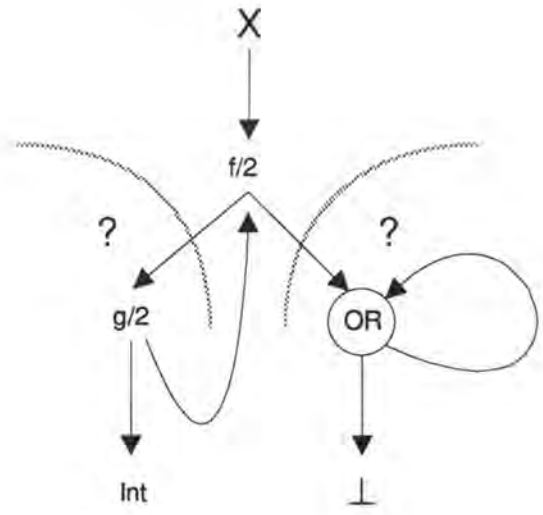


***Figure 5.8** : ID(f/2) depends on ID(g/2) and on ID(OR)*

When evaluating the functor "g" of the type-graph associated with the variable X, we could not compute its denotation because of its backward arc. We could only say that, from now on, we are not able to approximate the denotation of the functor "g" because one of its sons (which is the functor "f"). This is due to the property that all the sons of a functor are mandatory to evaluate its denotation.

It will be possible to give an approximation of the denotation of the functor "g" once we will know the denotation of the second son of the functor "f". That is made in the next paragraph.

As it can be shown in the following figure, we can immediately suppress the backward arc of the OR-node (1) because it adds no information to the denotation. By means of this deletion, the OR-node has become superfluous too because it rests only one alternative. This node could be suppressed too (2).

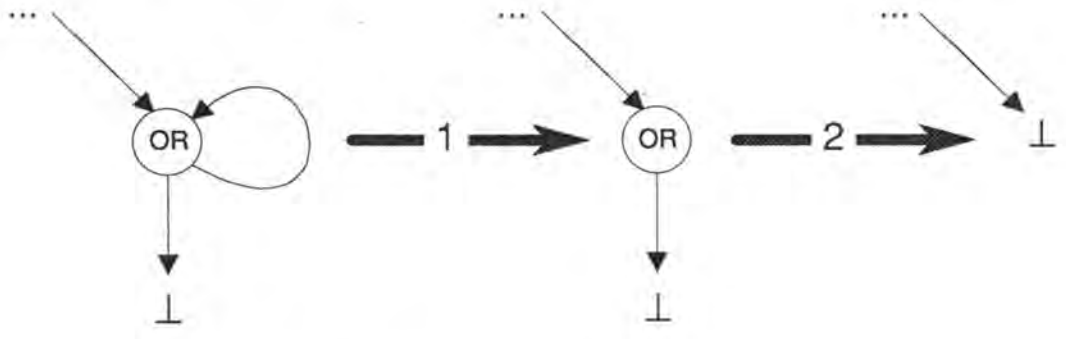


***Figure 5.9** : Reduction of an OR-node*

At this point, we can make two findings (illustrated by the next figure):

- as the denotation of the second son of the root is ⊥ and as the root is a functor, we can infer that the overall denotation will be ⊥. If the root was not a functor node but a

OR-node, the situation changes, indeed, all the sons has to be ⊥ to propagate this denotation at the upper level.

- as we knows the denotation of the second son of the root, we knows that the denotation of the first node is ⊥ too. As we already knows that the second son of the functor "f" is ⊥, this information is not necessary.

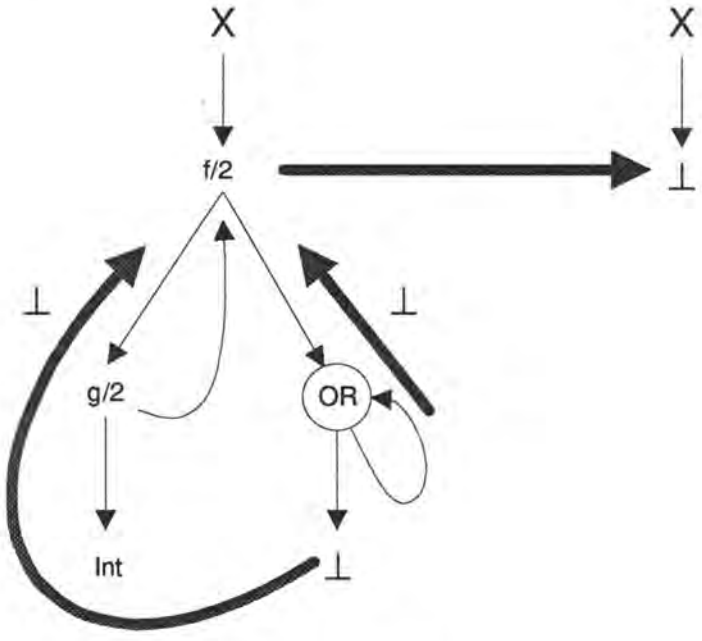
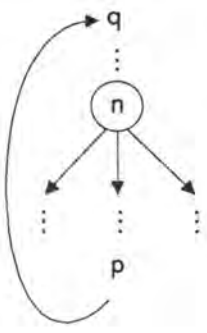


***Figure 5.10** : Upgoing of ⊥*

As shown in the previous example, the problems arise when we have to treat a node n which possess a descendant from which start backward arcs (p,q) such as Color(q) < Color(n).



This situation does not cause mandatory an unpredictable situation, it depends of the type of the node n.

If lb(n) = OR, it suffices to find a descendant of the node p which possess a n-node different of ⊥, to be sure to have a finite term.

If lb(n) = F/k, it suffices to have a backward arc higher than n to be in a unpredictable situation and this despite the denotation of the other sons (behalf ⊥ of course).

5.3.2. Formalization of non_empty_ID

Before specifying non_empty_ID, we need a boolean function Higher_Than_Backward which receives a node n as entry and which returns True, if n is higher that all the backward arc generated by its descendant.

Function Higher_Than_Backward(n)

PreCondition : <None>

PostCondition :

- This function returns True $\Leftrightarrow \exists m \in Desc^{+}(n): Color(m) < Color(n)$

Function Non_Empty_ID(node);

This function returns no more a boolean but can have three different values.

PreCondition : <None>

PostCondition :

- Non_empty_ID(n) is true if ID(n) is a non-empty set of finite terms.

Structure of the algorithm :

Base :

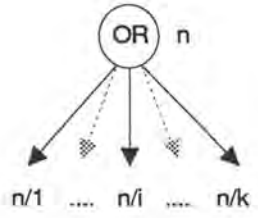
if lb(n) = $\bot$
then $\bot$
else $\neg\bot$

Step :

Case of the OR-node :

Suppose n to be a OR-Node, suppose k to be the number of its sons,

Non_Empty_ID(n) returns :

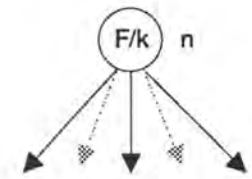


$$\left\|\begin{array}{l} \bot \Leftrightarrow \forall i\ (1 \le i \le k) : Non\_Empty\_ID(n/i) = \bot \\ ? \Leftrightarrow \forall i\ (1 \le i \le k) : Higher\_Than\_Backward(n/i) = False \\ \qquad\qquad\qquad\qquad \vee\ Non\_Empty\_ID(n/i) = ? \\ \neg\bot \Leftrightarrow \exists i\ (1 \le i \le k) : Non\_Empty\_ID(n/i) \neq \bot \end{array}\right.$$

Case of the Functor-node :

Suppose n to be a functor-node, suppose k to be the number of its sons,

Non_Empty_ID(n) returns :



$$\bot \Leftrightarrow \exists i\,(1 \le i \le k) : \text{Non\_Empty\_ID}(n/i) = \bot$$
$$? \Leftrightarrow \exists i\,(1 \le i \le k) : \text{Higher\_Than\_Backward}(n/i) = \text{False} \vee \text{Non\_Empty\_ID}(n/i) = ?$$
$$\neg\bot \Leftrightarrow \forall i\,(1 \le i \le k) : \text{Non\_Empty\_ID}(n/i) \neq \bot \wedge \text{Higher\_Than\_Backward}(n/i) = \text{Tr}$$

5.3.3. Completing the first pass

From this point, all the problems involved by the procedure non_empty_ID are resolved. We point out that this procedure had to make a travel through all the branches of the tree to up load the information. So, we will serve of this travel to bring up more information. We change a little bit Non_Empy_ID(n) so that it can returns the following values :

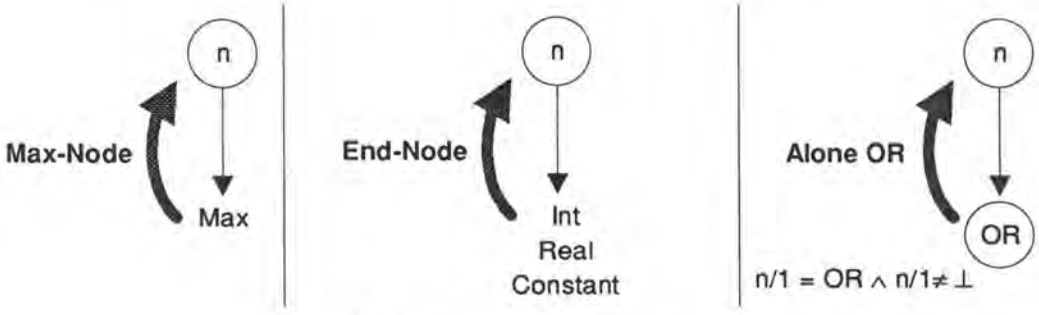


***Figure 5.11** : Extension of Non_Empty_ID*

From now on we have all the tools permitting to detect which rule to apply from 1, 2, 3, 4, 5 and 7. The implementation of the effect of those rules is not described here because it is nearly straightforward.

5.3.4. Focusing on the second pass

This pass has to realize only two rules.

5.3.4.1. Rule 6

> If a forward arc connects two OR-nodes (n,m) and indegree(m) = 1, then all arcs(m,k) are replaced by arcs (n,k) and the arcs (n,m) and the node m are removed.

This rule is straightforward to implement, it suffices to check when we visit an OR-node if one of its sons is an OR-node too and to kill this second OR-node if it has only one son.

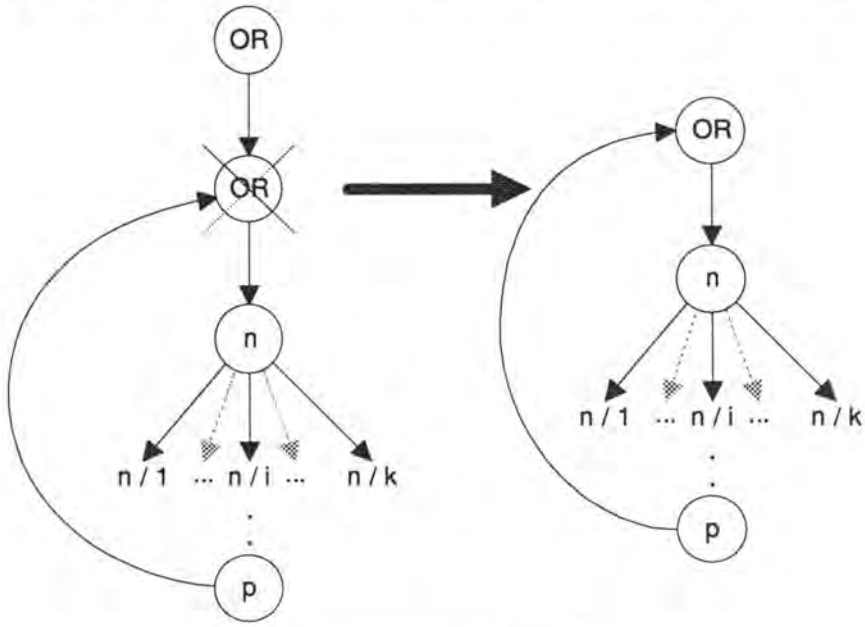


***Figure 5.12** : Two consecutive OR-node*

The only trick is to re-target the eventual backward arcs which were pointing to the suppressed node as expressed by the previous example. This can be easily made by means of our data structure, it suffices to found in all descendants of the second OR-node which have the same field *Info* that the suppressed OR-node and replace the reference to it by a reference to the first OR-node.

5.3.4.2. Rule 8

> If there is a backward arc (n,m) and the forward path m, k_1, ..., k_n with k_n = n constits of OR-nodes then all backward arcs (1,k_i) are replaced by the backwards arcs (1, m).

This rule causes no particular difficulties. It suffices to create a stack which allows to memorize the path taken to reach a node. During the visit of a OR-node, when a backward

node is found, it suffices to consult in the stack to know if a chain of OR-node link the target of the backward arc to it destination, as it can be shown in the next figure.

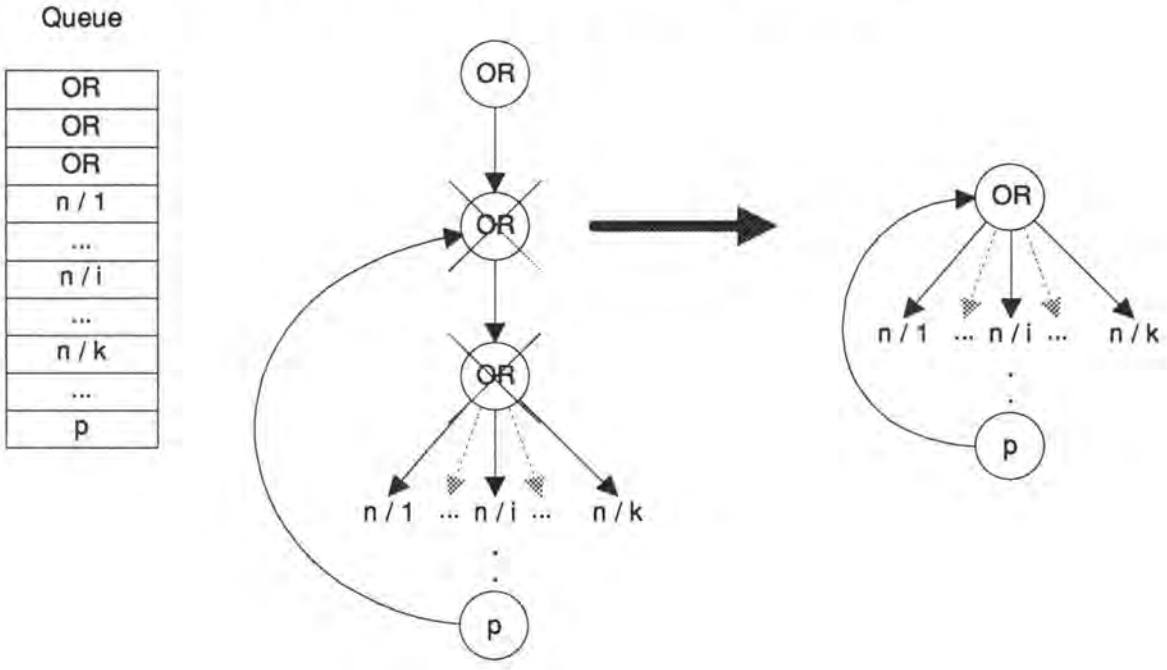


Figure 5.13 : A chain of OR-node

If it is the case, all the sons on the path OR-node are attached on the first node off the path and all the other are removed. The problem of re-targeting the eventual backward arcs which were pointing to the suppressed node arises again, the same solution as for the previous rule can be employed.

5.4. Restricted type-graphs

As it can be observed when looking at the algorithm Restrict, the proposed specification is very complete and impressive. So, we decide to implement this algorithm like proposed in the thesis.

The first task was to conceive all the procedures appearing in this algorithm. The second task was to found the data structure which can map as perfectly as possible to functions (nd and nfr) used in the algorithm, it was done by means of single chained list. All the information needed by this algorithm is stored in a stack which is principally used by the function Safe_Anc.

Explaining more this algorithm is redundant with the explanation given in the thesis, so we decide to stop this section here.

5.5. *Other Algorithms*

Those algorithms are divided in two groups. The first group returns a boolean which is the result of the test constituting the algorithm. The second group receives one or two type-graphs as input and returns a new type-graph which is the result on an operation defined by the algorithm.

The first group is composed of the two following algorithms :
- Equivalence(n1,n2) returns True if ID(n1) ≡ID(n2) otherwise False,
- Smaller_Or_Equal(n1,n2) returns True if ID(n1) $\subseteq$ ID(n2) otherwise False.

Both algorithms of this group presented in the thesis have nearly the same structure, and use at any rate the same technique. They use a set to memorize the path taken to come down a node. For example, the first line of both algorithms presented are "if (n,m) $\in S^c$ then ...".We have to build a data structure corresponding to this set, we choose a stack to do this. This stack will be used only to store the pairs of nodes by which the algorithm pass. This stack always grows, no elements are removed from it because the set S^c never shrinks.

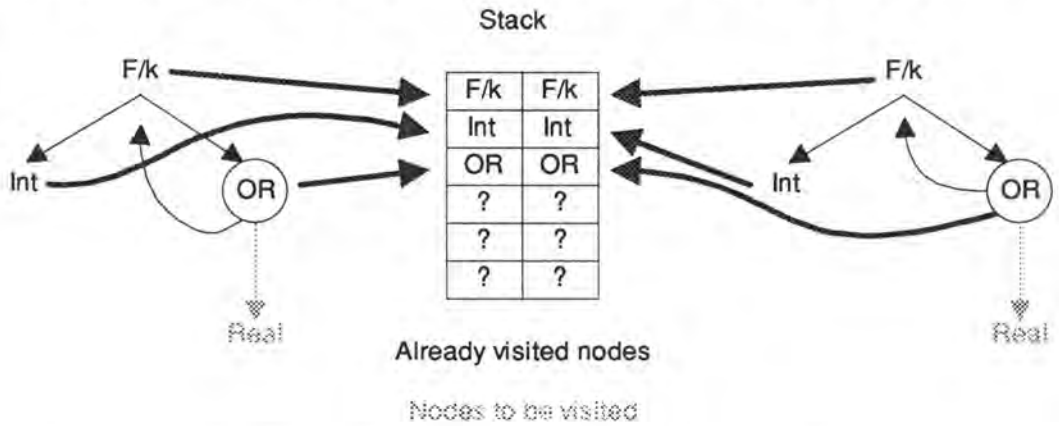


***Figure 5.14** : Implementation of the stack*

The implementation of the set S^c was the main difficulty for the algorithms of this group, the stack presented here permits to sketch it perfectly. The other problems encountered during the implementation of those algorithms are traditional in the sense that they presents any specific difficulties, so we will pass directly to the other group.

The second group is composed of the following algorithms :
- T_c = Compact(T_c) which returns a compacted type-graph,
- T_r = Restrict(T_r) which returns a restricted type-graph,
- T_{12} = Intersection(T_1,T_2) which returns the intersection on both type-graph,
- T_{12} = Bunif(T_1,T_2) which returns the backward unification on both type-graph (its exactly the same operation than intersection).

Two of the algorithm of this group were discussed above, so we will just evoke the algorithm Intersection here. This algorithm is directly derived from Restrict and we adopt the same manner to solve it. First, we have conceived all the procedures appearing in this algorithm. Afterwards, we have chosen the data structure which can map as perfectly as

possible functions (*is* and *ris*) used in the algorithm, it was done like in Restrict, by means of single chained list. All the informations needed by this algorithm are stored in a stack which is principally used by the function Safe_Anc.

Once the algorithm implemented, we have tested this algorithm with a lot of examples. In general, this algorithm gave the intended result, but sometimes, it returns us a bad response. This was due to our bad understanding of the point 4 (illustrated next) of the algorithm.

$$\begin{array}{l}\text{else } \% \ \exists n \in ris(l): lb(n) = f/k \text{ or } lb(n) = Int, Real..... \% \\ \quad \text{if } \forall m \in ris(l) \setminus \{n\}: (\exists m_d \in pmd(m): lb(m_d) = lb(n)) \\ \quad \text{then } lb(l) \leftarrow lb(n),\ S^{ul} \leftarrow S^{ul} \setminus \{l\}, S^{sn} \leftarrow S^{sn} \bigcup \{l\}, \\ \qquad \{\cdots\} \\ \quad \text{else } lb(l) \leftarrow \perp,\ S^{ul} \leftarrow S^{ul} \setminus \{l\}, S^{sn} \leftarrow S^{sn} \bigcup \{l\}\end{array}$$

Our algorithm was, as declared above, an exact translation of the proposed one. It contains the same functions (*ris*, *is*) under form of chain list. We can found in the thesis, that a node l of T_{12} has an associated function *is*. Its value, *is(l)*, is a subset of $DESC^*(n1) \cup DESC^*(n2)$. In fact, *is(l)* always contains one or two nodes. The bug in our program come from the use of this function.

Suppose that ID(*is(l)*) = {Int,Max}, in this case, ID(*ris(l)*) = {Int} because *ris(l)* is the set $\{n \mid n \in is(l) \text{ and } lb(n) \neq Max\}$. And finally, suppose that we must perform the point 4 of the previous algorithm.

Following the algorithm, it exists an element in *ris(l)* such as its label can be (f/k, Int, Real). In the previous example, lb(n) = {Int}, this is the only possibility because *ris(l)* contains only one element.

Once n chosen, we have to evaluate the following condition :

$$\forall m \in ris(l) \setminus \{n\}: (\exists m_d \in pmd(m): lb(m_d) = lb(n))$$

In this case, $ris(l) \setminus \{n\} = \varnothing$ then the condition is always satisfied. So, the second part of the condition must not be evaluated. During the implementation, this case must be take into account. Because if it was not, the algorithm does not respect its specification.

This situation shown the weakness of a programming language comparing to mathematical notation. Because, of this, the implementation must be very pragmatic.

5.6. *Summary*

In this chapter, we have presented all the data structures chosen to represent the type-graphs. It follows a new version of the algorithm Compact which is a radically different algorithm than the one proposed in the thesis.

The section 5.4 describes the implementation of Restrict and is prolonged by an overwiew of the main problems appeared during the implementation of the algorithm Equivalence, Smaller_Or_Equal and Intersection.

Chapter 6

Implementation of the same-value and of the abstract substitutions

6.1. Introduction

We will begin section 6.2 by a description of the data structures used to implement the same-value component belonging to the abstract domain. The previous data structure implies some fundamental changes in the representation of this SVAL-component. The motivation which conducted to those major changes are also presented in the same section.

The section 6.3 contains a complete description of the data structure created to denotes the substitutions.

The next section contains a specification of the algorithm which construct the SVAL-component corresponding to the chosen data structure.

The section 6.5, is an overview of differents algorithm's working on the substitutions. It contains a short descrition of the tricks imposed by their structure and an overlook at the solutions proposed.

Finally the section 6.6 contain an overview of coarse problems derived form the data structure as of the solutions.

6.2. Representation of the same-value

Definition 6.1.

Let n be a node in the type-graph. n/ε selects the node n itself.
$n/(f_1/k_1,i_1).\cdots.(f_{p-1}/k_{p-1},i_{p-1}).(f_p/k_p,i_p)$ with $1\leq i_1\leq k_1$, ... , $1\leq i_p\leq k_p$ selects the i_p^{th} son of the principal node with label f_p/k_p of the node $n/(f_1/k_1,i_1).\cdots.(f_{p-1}/k_{p-1},i_{p-1}).(f_p/k_p,i_p)$.

We called ε and $n/(f_1/k_1,i_1).\cdots.(f_{p-1}/k_{p-1},i_{p-1}).(f_p/k_p,i_p)$ selectors. Selectors are denoted by the letters s and p possibly with a superscript or a subscript.

SVAL-constraints are defined as couples of selectors on two variables. An SVAL-constraint has the form $\{X/s_X, Y/s_Y\}$ with s_X and s_Y determinate selectors in respectively T_X^β and T_Y^β, where T_X^β is the type-graph associated with the variable X in the abstract substitution β and where T_Y^β, is the type-graph associated with the variable Y in the same abstract

substitution. This constraint expresses the fact that $X\theta/s_X = Y\theta/s_Y$ must hold in any concrete substitution Θ represented by β.

The representation of the same-value presented in the thesis induces a couple structure representation, this couldn't be the best way in the perspective of an implementation, as we will show it after. So, we decide to extend the couple representation $\{X/s_X, Y/s_Y\}$ to a n-tupple representation. A SVAL-constraint on variables $\{X_1,..., X_n\}$ of an abstract substitution β will be expressed like a set of variables associated with a selector $\{X_1/s_{X_1},...,X_n/s_{X_n}\}$. This modification will not distress so much the proof presented in the thesis because it suffice not to consider a SVAL-constraint like a set of couple of variables but like a set of variables. In the place the authors where using a set of couple on variables, we use only a set of variables.

6.2.1. Representation of the SVAL-constraint

Definition 6.2.

An SVAL-constraint has the form $\{X_1/s_{X_1},...,X_n/s_{X_n}\}$ with $s_{x_1},..,s_{x_n}$ are determinate selectors in respectively $T^{\beta}_{X_1},..,T^{\beta}_{X_n}$, where $T^{\beta}_{X_1}$ is the type-graph associated with the variable X_1 in the abstract substitution β,..., and where $T^{\beta}_{X_n}$ is the type-graph associated with the variable X_n in the same abstract substitution. This constraint expresses the fact that $X\theta_1/s_{x_1}$ =...= $X\theta_n/s_{x_n}$ must hold in any concrete substitution θ represented by β. Several reasons motive those changes, we will show it by means of the example above.

Example

Let X_1, X_2, X_3 be three variables on the abstract substitution β, and suppose that there is a same-value between the root of X_1 and X_2 $\{X_1/\varepsilon_{X_1}, X_2/\varepsilon_{X_2}\}$ and that there is a same-value between the root of x_2 and $x_3\{X_2/\varepsilon_{X_2}, X_3/\varepsilon_{X_3}\}$.

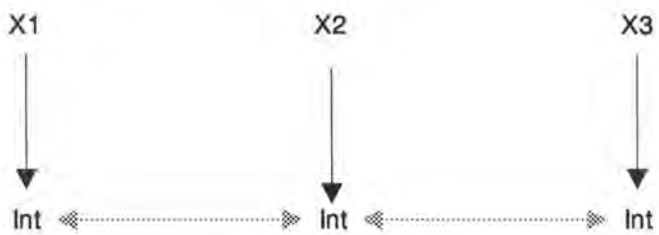


***Figure 6.1** : Two same-value constraints*

With the representation proposed in the thesis, this SVAL-constraint must be completed by a new couple $\{X_1/\varepsilon_{X_1}, X_3/\varepsilon_{X_3}\}$. This operation of extension is not realised at the time the second constraint was added but during an operation of extension performed by the algorithm Normalize.

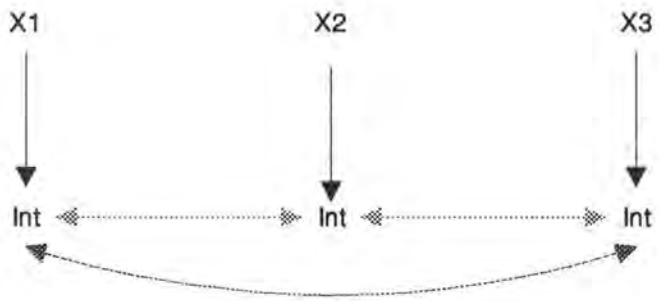


***Figure 6.2** : A new same-value constraint added by transitivity*

This representation suffers of a big inconvenient, each selector has to be represented more than once when there is a SVAL-constraint added because of the transitivity of relation "same-value". In the simple example above, each selector is represented twice and adding a same value on one of those tree variables with a fourth one creates two new constraints by transitivity for a total of six.

$$\left\{\begin{array}{l}\{X_1/\varepsilon_{X_1}, X_2/\varepsilon_{X_2}\}, \{X_2/\varepsilon_{X_2}, X_3/\varepsilon_{X_3}\}, \{X_1/\varepsilon_{X_1}, X_3/\varepsilon_{X_3}\}, \\ \{X_1/\varepsilon_{X_1}, X_4/\varepsilon_{X_4}\}, \{X_2/\varepsilon_{X_2}, X_4/\varepsilon_{X_4}\}, \{X_3/\varepsilon_{X_3}, X_4/\varepsilon_{X_4}\}\end{array}\right\}$$

It is easy to see that when the number of SVAL-constraints grows up, this representation is not viable so, we decide to adopt the n-tupple representation which is more concise and less memory consuming than the other one. Adapted on the previous example, the n-tupple representation will give :

$$\{X_1/\varepsilon_{X_1}, X_2/\varepsilon_{X_2}, X_3/\varepsilon_{X_3}, X_4/\varepsilon_{X_4}\}$$

You can remark that all the selectors are represented once and that there is no longer redundancies in the notation. It is straightforward to see that all the operations that could be done on a couple can be done on the n-tupple too. This is due to the fact that the principal property of the pairs i.e. containing only two elements, is never used.

Shortening the notation and minimizing the memory space was not the alone purpose of this representation, an overlook to the algorithm Normalize on abstract substitutions teach us that the normalization can be very time-consuming. The authors propose the algorithm like an operation which receive an abstract substitution as input and which return as output an abstract substitution respecting the clauses defined by the algorithm. Due to the prolog-like specification of this algorithm and to the chosen representation, the transformation of the abstract substitution doesn't seem to be very hard to do. But with a imperative language like C, the transformation will be intricate, so we decide to abandon the normalize operation defined in the thesis and to choose another way to proceed. Our aim is to always have a normalized abstract substitution and thus avoid the Normalize operation. This can be done by means of a powerful procedure which detects during the addition of a new SVAL-constraint any infraction to the rules presented in the thesis and which can react to this infraction. This algorithm is presented later but now, we have to create the data structures corresponding to the new representation of the SVAL-constraint proposed above.

The problem is to indicate how much variables are bound together and to represent the selector itself. The general thought is to implement the selector like a tree starting from the root of the type-graph of a variable and stopping at the place where the SVAL-constraint stands. The second idea was to merge the "end-node" of each selector in an unique node and to create a reference to the path leading to this merged node. A representation of SVAL-constraint if shown in the next figure.

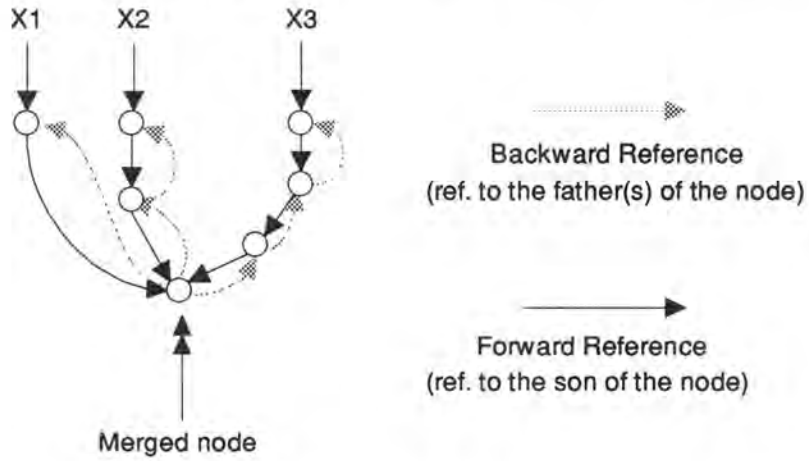


***Figure 6.3** : A merged same-value constraint*

We have 3 generic types of node :

1. the node denoting a functor,
2. the node denoting father of the functor node,
3. the node denoting sons of the functor node.

Cell representing a node denoting functor

ColorSelect	Nbsharing	ListFather	SonSelect

where :

ColorSelect is an integer which will be used for "colouring" the node. This technique permit, as in the type-graph, to memorise if we already have visited a node during the traversal of the SVAL-constraint.

NbSharing is an integer used to indicate how much time the functor node is referenced. This information will be used at the desallocation time to know if we can destroy a node or not (see later for more).

ListFather is a pointer to a single linked chain of node representing the father of the functor node.

SonSelect is a pointer to a single linked chain of node representing the sons of the functor node.

Cell representing a node denoting the father of a functor node

NoVar	FatherSelect	NextFather

where :

NoVar is an integer which reference a number of variable. This variable <NoVar> has a same-value constraint which lead to the functor node from which start the list of father containing this cell.

FatherSelect is a pointer to the functor node father of the node from which start the list of father containing this cell.

NextFather is a pointer to the next father of the list.

Cell representing a node denoting the sons of a functor node

NoSon	NodeSelect	NextSelect

where :

NoSon is an integer which references son's number. Contrary to the type-graphs, the list of sons contains not all the sons of the functor node but just the sons which lead to a descendant on which there is a same value.

NodeSelect is a pointer to a functor node which leads to (or which is) a SVAL node[4].

NextSelect is a pointer to the next son of the list.

Example :

Let X_1, X_2, X_3 be three variables on an abstract substitution β. We want to indicate that there is a same value constraint on those three variables with respectively the selector $\{X_1/1, X_2/\varepsilon, X_3/1\}$. This configuration can be represented by those figures below.

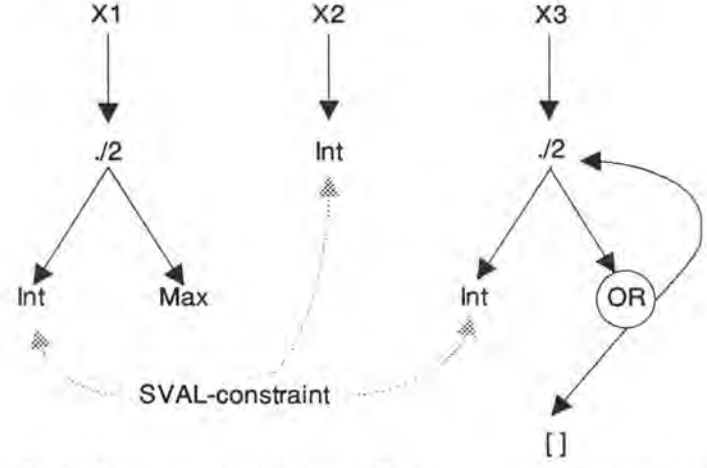


Figure 6.4 : *Representation of the Sval-constraint{X_1/1,X_2/ε,X_3/1}*

[4] Notation: we called an SVAL-node the "end-node" of a selector.

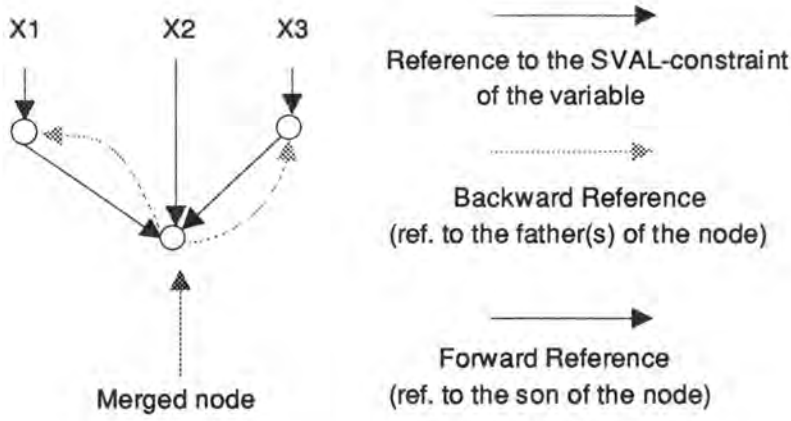


***Figure 6.5** : Another representation of the Sval-constraint{X_1/1,X_2/ε,X_3/1}*

Our data structure permits to have all the selectors associated with the corresponding variables in one shot because of the list of father's attached to each functor node. This SVAL-constraint can be denoted by our data structure like in the next figure, the list of son has been removed for reasons of readability.

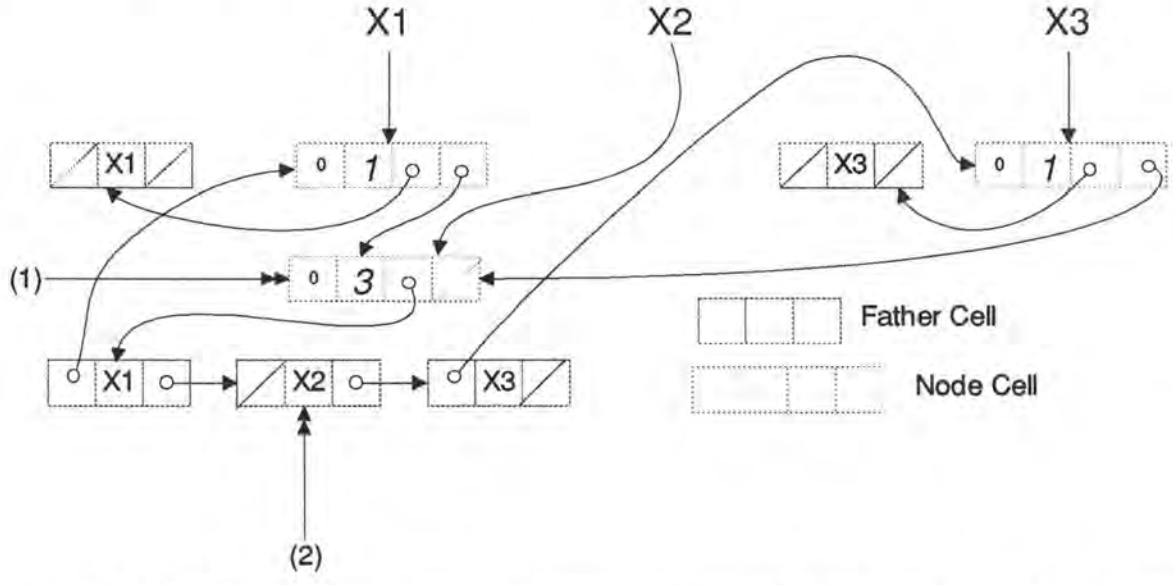


***Figure 6.6** : Data structure chosen for representing the Sval-constraint{X_1/1,X_2/ε,X_3/1}*

It can be observed that the data structure can become very intricate when the number of variables affected by an SVAL-constraint grows. In the previous figure, we can point out some particularities :

- The three variables share a same functor node (1) but at different depth. This node is the son of the root of the variable X_1, is in the same time the root of the variable X_2, and finally this node is the son of the root of the variable X_3. This is why the fields *NbSharing* is set to the value three. We can easily reconstruct the selector leading to the merged node. It suffice to start from the root of a variable, travelling through the tree by the sons of the functor node till reaching the SVAL-node.

- The father node (2) has no father, this means that the functor node from which starts the list of father containing this father node is a root.
- The variable's number is always presents in the list of father, this is permit to know when we are on a SVAL-node which are the variables involved by this SVAL-constraint.
- The field *ColorSelect* does not serve in this example so, it is set to a null value.

We have defined how to represent a SVAL-constraint on a set of variables but practically, we can have several SVAL-constraints on the same variable. We show in the next section how to use our data structure to face this situation.

6.2.2. Representation of the SVAL-component

To represent a SVAL-component on a variable, we can keep the same data structure. The only change is that a SVAL-node is not always an "end-node", it can lead to reach an other SVAL-node.

Example :

Let X_1, X_2, X_3 be three variables on an abstract substitution β. Suppose that there is a SVAL-constraint on $X_1, X_2 : \{X_1/1, X_2/1\}$.

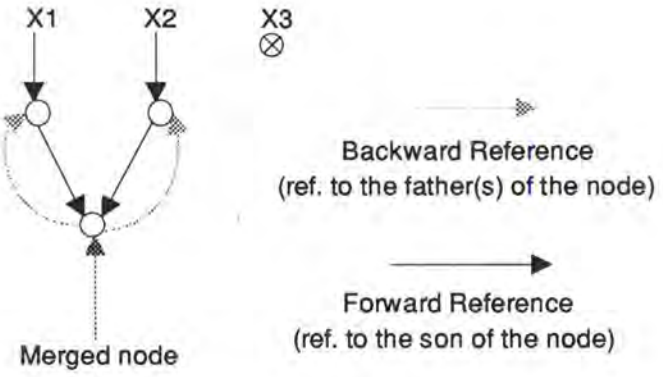


***Figure 6.7** : A simple Sval-constraint*

And now, suppose that there is a SVAL-constraint on X_1, X_3 $\{X_1/(1,1), X_3/(1,1)\}$.

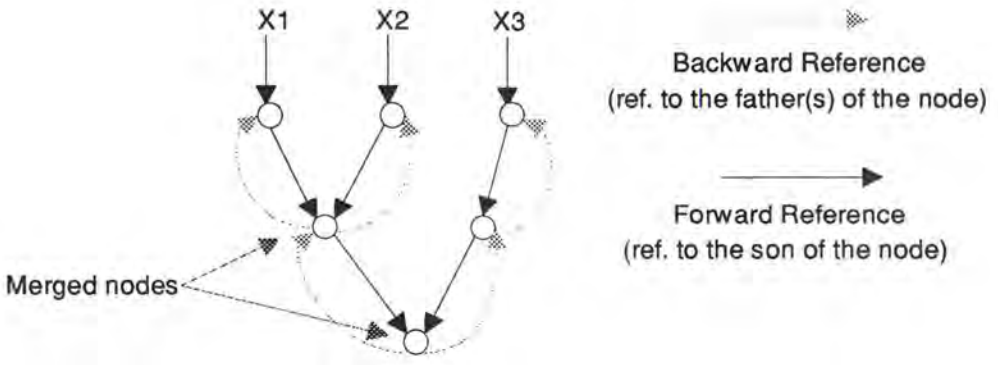


***Figure 6.8** : A more complex Sval-constraint added to a simple one*

As shown in the previous figures, we use the path to the first merged node to create the path leading to the second one. Obviously, this can be done only if the paths have an identical portion in common as it is the case here (the second selector extends the first one). If it is not the case, it suffices to add the selector to the root as if there was no same-value (creating a new son to the root).

From now on, we have defined the representation of the SVAL-constraints, in the next section, we present the data structures chosen to represent an abstract substitution.

6.3. *Representation of the abstract substitutions*

Before introducing the data structures chosen to represent the abstract substitutions, we shall make some general considerations on the same-value like proposed in [7]. In this work, the SVAL-component refers always to a corresponding type-graph. It can be easily seen by means of the notation $\{X/s_X, Y/s_Y\}$. The selector s_X define a path which must maps the type-graph. They use always type-graph that are well defined for each step of the selection in the sense that there always exists an appropriate principal node or term to be selected, as expressed in the next figure.

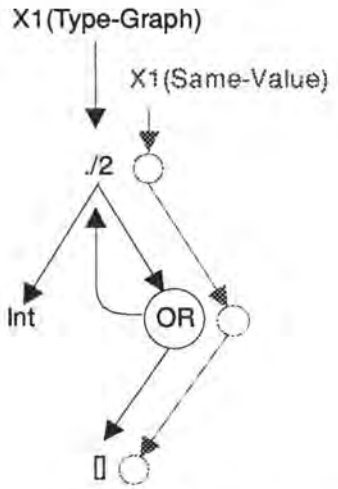


***Figure 6.9** : Correspondance between the Sval-constraint and the Type-graph*

Meanwhile, the same-value composent can be expressed without the help of the type-graphs. A lot of the abstract domain defined in the litterature possess a same-value component without type-graph. In this case, an other technique replace the mechanism of selection used here.

Because of the close relation getween the type-graphs and the SVAL-component in this domain, an abstract substitution is defined as follow (this definition is expressed in our formalism).

Definition 6.3 :
An abstract substitution β over a domain $D = \{X_1,...,X_i,...X_n\}$ is either $\perp$ or a pair (type,sval).

The *TYPE-component*, type, associates a normal rigid type-graph with each variable in the domain D and its selected by TYPE(β).
TYPE(β) = {X←T_X | X ∈ D and T_X is a non-empty normal rigid type-graph}.

The *SVAL-component*, sval, is a set of *SVAL-constraints* and is selected by SVAL(β). A SVAL-constraint has the form $\{X_1/s_{X_1},\ldots,X_n/s_{X_n}\}$ with $s_{x_1},..,s_{x_n}$ are determinate selectors in respectively $T^{\beta}_{X_1},..,T^{\beta}_{X_n}$, where $T^{\beta}_{X_1}$ is the type-graph associated with the variable X_1 in the abstract substitution β,..., and where $T^{\beta}_{X_n}$ is the type-graph associated with the variable X_n in the same abstract substitution. This constraint expresses the fact that $X\theta_1/s_{x_1}$ =...= $X\theta_n/s_{x_n}$ must hold in any concrete substitution θ represented by β.

By convention an abstract substitution β is denoted by ⊥, if at leads one of the types $T^{\beta}_{X_i}$ is a ⊥-node. This abstract substitution will be called later minimal abstract substitution. In the same way, an abstract substitution β is denoted by Top, if all of the types $T^{\beta}_{X_i}$ are a Max node. This abstract substitution will be called later maximal abstract substitution.

Construct a data structure permitting to maps the previous definition of the abstract substitution is quite easy because an abstract substitution is just composed of a list of variables $\{X_1,...,X_i,...X_n\}$ containing references to the type-graph $(T^{\beta}_{X_1},..,T^{\beta}_{X_n})$ and same-value $\{\{X_{1_1}/s_{X_{1_1}},\ldots,X_{n_1}/s_{X_{n_1}}\},\ldots,\{X_{1_k}/s_{X_{1_k}},\ldots,X_{n_k}/s_{X_{n_k}}\}\}$ associated to each variable belonging to the abstract substitution. The next figure give an example of abstract substitution as defined above. This example is the graphical representation of the following abstract substitution.

β ≡ {X_1,X_2, X_3}
where TYPE(β) = {X_1← f(Real,g(Int)), X_2← k(g(Int)), X_3←Int}
SVAL(β) = {{X_1/2, X_2/1},{X_1/(2,1), X_2/(1,1), X_3/0}}

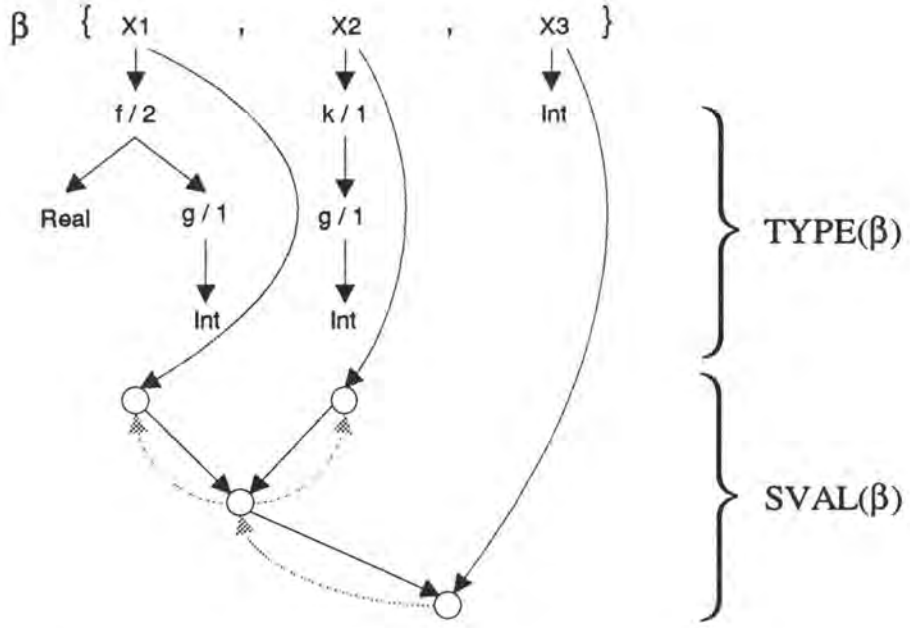


Figure 6.10 : *Complete representation of a substitution*

We shall no define more formally the data structure chose to represent the abstract substitution. The traditional single linked chain suits perfectly the list of variables.

Cell representing a variable in the domain

NoVariable	TypeGraph	SameVal	SValOnRoot	NextVar

where :

NoVariable is the reference number of the variable in the abstract substitution.
TypeGraph is pointer to the type-graph associated to variable <*NoVariable*>.
SameVal is pointer to the SVAL-constraints which implies variable <*NoVariable*>.
SvalOnRoot is a flag indicating if there is a SVAL-constraint on the root of the same-value associated to variable <*NoVariable*>.
NextVar is a pointer to the next variable of the list.

We choose a more developed structure to represent the abstract substitution than only the single linked chain. We have creates a record structure containing four fields as described below.

Cell representing an abstract substitution

where :

NbVariable is the number of variables present in the abstract substitution.
TopSubst is flag indicating if the abstract substitution is the maximal abstract substitution.
BottomSubst is flag indicating if the abstract substitution is the minimal abstract substitution.
SonSelect is a pointer to the single linked chain of variables.

From now on, we know how to represent the SVAL-constraints composing the SVAL-component, we know how to represent the abstract substitution, we must now create the operations using those data structures, and particularly, we are focusing on the Normalize operation.

6.4. *Normalization of an abstract substitution*

As explained before, the aim of our representation of the SVAL-component is to avoid the operation of normalization as presented in the thesis. We decide to always have normalized abstract substitutions, so we must take care when adding component to keep a normalized representation.

Now, we shall focus on the algorithm Normalize as described in the thesis, and examine it rule by rule to state on the changes brought out by our representation of the data structure.

6.4.1. Algorithm Normalize

This algorithm is based on the following considerations which can be found in [7]. The type being the same, there are many SVAL-component that yield the same $\gamma(\beta)$. One of the reasons is that equality is a transitive relation. If, for example, $\{X/s_X, Y/s_y\}$ and $\{Y/s_y, Z/s_Z.s\}$ belong to SVAL(β), then $X\theta/s_X = Z\theta/s_Z.s$ for all $\theta \in \gamma(\beta)$. So, adding $\{X/s_X, Z/s_Z.s\}$ to SVAL(β) does not change the denotation. However, its presence is desirable to express the dependencies between X and Z. On the other hand, $X/s_X = Y/s_y$ implies also $X/s_X.s = Y/s_y.s$. Presence of the latter is undesirable because it only enlarges the SVAL-component. It is preferable to have the SVAL-component in a form where constrains between any pair of variables are at the same time explicit and minimal, and to have the TYPE component in a form

that is compatible with the SVAL-component, in the sense that if $\{X/s_X, Y/s_y\} \in SVAL(\beta)$ then $n_0^{X\beta} / s_X \equiv n_0^{Y\beta} / s_Y$.

6.4.1.1. Rule 1

> if $(\{X/s_X, Y/s_y\}, \{X/s_X.s, Z/s_z\} \in SVAL(\beta_n)$ and
> $n_0^X / s_X \equiv n_0^Y / s_Y$ and $n_0^X / s_X.s \equiv n_0^Z / s_Z$ and
> not $(\exists \{Y/p_y, Z/p_z.s\} \in SVAL(\beta_n)\ \exists\ p : p_y.p = s_y.s,\ p_z.p = s_z))$
> then $\{Y/s_v.s, Z/s_z.s\}$ is added to $SVAL(\beta_n)$

This rule has been created to take into account transitivity of the SVAL-constraint. It only adds a new SVAL-constraint if the existing constraints are compatible with the types and if the new constraint is not less restrictive than the existing one. So, due to the existing constraints and due to the conditions imposed on the types, the new SVAL-constraint does not affect $\gamma(\beta)$. By means of our representation of the SVAL-constraint, this first rule is no more useful. Our representation contains automatically all the constraints added by transitivity and the less restricitive constraint are removed before the addition of a new one (see later for more).

6.4.1.2. Rule 2

> if $(lb(n_0^X / s_X) = lb(n_0^Y / s_Y) = f/k$ and
> $\forall\ i \in [1,k] : \{X/s_X.i, Y/s_y.i\} \in SVAL(\beta_n)$ and $n_0^X / s_X.i \equiv n_0^Y / s_Y.i)$
> then $\{X/s_X, Y/s_v\}$ is added to $SVAL(\beta_n)$

This rule makes an SVAL-constraint explicit that exists implicitly. Hence, $\gamma(\beta)$ the concretization of β is not changed.

This rule has to be implemented if we consider that the SVAL-component must be minimal. But it is not a mandatory for the correctness of abstract interpretation.

6.4.1.3. Rule 3

> if $(\{X/s_X, Y/s_y\}, \{X/s_X.s, Y/s_y.s\} \in SVAL(\beta_n)$ and $s \neq \varepsilon)$
> then the latter element is removed from $\in SVAL(\beta_n)$

This rule has been created to remove a SVAL-constraint which can be subsumed by another SVAL-constraint. By subsumtion, we means that the selector of an SVAL-constraint $\{X/s_X, Y/s_y\}$ is longer than the selector representing another SVAL-constraint $\{X/p_X, Y/p_y\}$. This does not affect $\gamma(\beta)$.

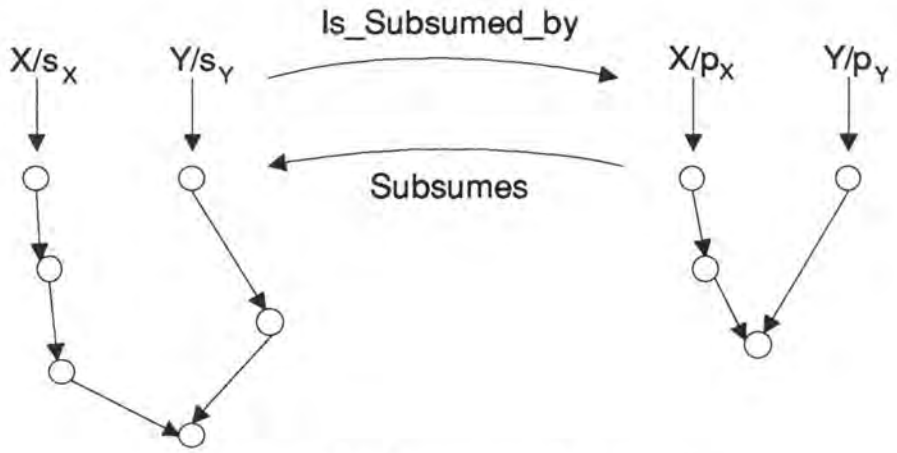


***Figure 6.11** : An example of subsumtion*

This rule can be avoided easily, it suffices to check the presence of such subsumtion before the addition of a new SVAL-constraint on an existing abstract substitution. If the new SVAL-constraint is subsumed by an existing one in the abstract substitution, we decide to not insert it in the abstract substitution because it is redundant. In the same way, if an existing SVAL-constraint is subsumed by the new SVAL-constraint, we decide to remove it from the existing abstract substitution.

6.4.1.4. Rule 4

if ($\{X/s_x, Y/s_v\} \in SVAL(\beta_n)$ and $s \neq \varepsilon$) then β_n becomes $\perp$

We assume here that we use a Prolog Compiler with occur check. The functionality of this rule is to check the presence of a loop in the same-value. If a loop is found this means that the denotation of a type-graph contains the type-graph itself. A term and one of its subterm has an identical value. As both selector are determinate, we know that all the terms in T_x^{β} have this circularities, and thus all are infinite terms. This situation can appear when a Prolog program can not terminate. For example, the following Prolog program always returns a failed answer (if the occur check is implemented in the compiler).

$$p(X_1,X_2) :- X_1 = f(X_2), X_2 = X_1.$$

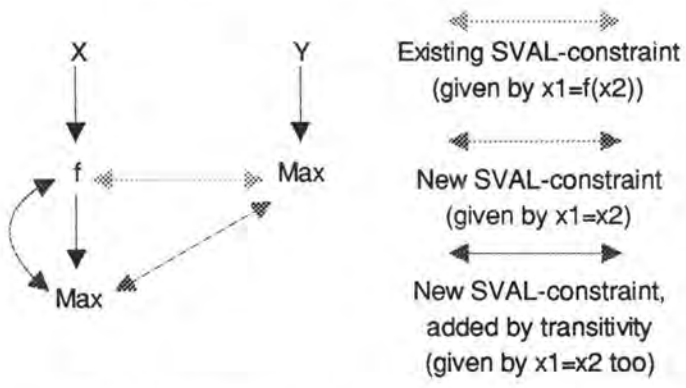


***Figure 6.12** : A looping Sval-constraint*

This situation can be expressed by our data structure, but as for the detection of the subset, we decide to check if the introduction of a new SVAL-constraint will generate this looping situation and to react by setting the abstract substitution to Bottom if it is necessary. In fact, $\gamma(\beta) = \varnothing = \gamma(\bot)$.

6.4.1.5. Rule 5

> if ($\{X/s_X, Y/s_y\} \in SVAL(\beta_n)$ and
> (if X=Y then s_X and s_y do not overlap) and not($n_0^X / s_X \equiv n_0^Y / s_Y$)
> then T = intersection(n_0^X / s_X, n_0^Y / s_Y) and
> $T_X^{\beta'} = replace(T_X^{\beta'}, s_X, T)$ and $T_Y^{\beta'} = replace(T_Y^{\beta'}, s_y, T)$
> and if ($lb(T_X^{\beta'}) = \bot$ or $lb(T_Y^{\beta'}) = \bot$) then β_n becomes $\bot$

In this rule, the SVAL-constraint $\{X/s_X, Y/s_y\}$ implies that in each concrete substitution $\theta \in \gamma(\beta)$, X^θ / s_x must be identical to Y^θ / s_y. Thus, only terms that are in $ID(n_0^X / s_X)$ and in $ID(n_0^X / s_X)$ are appropriate, all other terms can never occur in any θ. We can as well remove those superfluous terms by replacing the sub-graphs by T = intersection($n_0^X / s_X, n_0^Y / s_Y$). We see that it does not change $\gamma(\beta)$. If T_X^β or T_Y^β is a $\bot - node$, then can β be changed into $\bot$ as $\gamma(\beta) = \varnothing = \gamma(\bot)$.

6.4.2. Informal algorithm

Because of all those changes, the implementation of our normalization algorithm is quite different of those proposed in the thesis. The major work is done when a new SVAL-constraint is added to the abstract substitution. This is due to the fact that we are always consider normalized abstract substitutions and that we keep always the abstract substitution in the same state. This implies a to perform a lot of test on the existing substitution and on the new SVAL-constraint to be added.

Before explaining informally our algorithm, we will describe how the new SVAL-constraint is stored before being integrated into the abstract substitution β. This new constraint result of the unification of two variables in the Prolog program. For example, the build-in "$X_1 = X_2$" add a new SVAL-constraint $\{X_1/\varepsilon, X_2/\varepsilon\}$ between the variables X_1 and X_2.

This constraint is like any other SVAL-constraint and thus can be represented on an abstract substitution δ, it has its proper TYPE-component and SVAL-component. The type-component is nearly the same than those presents in TYPE(β). The only change is on the nodes where the selectors of SVAL(δ) lead. As δ contains only one SVAL-constraint as SVAL-component, it suffices that it respect the rules 3 and 5 to be normalized. The application of thoses rules to the abstract normalized substitution δ does not cause problem as it exists no other SVAL-contraint on SVAL(δ).

Adding the abstract substitution δ to the abstract substitution β is like doing the unification of both substitutions. So, instead of an unique parameter β as the algorithm presented in the thesis, our algorithm normalize has two parameter β_{in} and δ where β_{in} is an abstract normalized substitution and δ an abstract substitution containing an unique SVAL-constraint.

This algorithm is presented informally here, a complete specification follows. This algorithm permit to add the SVAL-constraint defined on δ onto the abstract substitution β. This algorithm will be called for each SVAL-constraints which has to be added.

```
Add_Constraint(δ,β )
        ↓
if (Is_Subsumed_By(δ,β ) == TRUE)
then { if (Subsume(δ,β ) == TRUE)
         {Kill_Subsumed(δ,β );}
       β = Add_Constraint_On_Existing_Subst(δ,β );
       β = StandardizeSubst(β);
else Nothing
```

It is very easy to detect if a SVAL-constraint is subsumed by another with our data structure. It suffice to come down in the tree representing the SVAL-component in β by the path defined in the SVAL-constraint of SVAL(δ). If we detect a node in SVAL(β) (before the end of the selector defined in SVAL(δ)) with a list of father which contains at least the same list of variables than those present at the SVAL-node of SVAL(δ), we can be sure that this SVAL-constraint is subsumed by the one already present in SVAL(β).

In the example explicated by the following figure, we can see that adding the new SVAL-constraint contain in SVAL(δ) will introduce a redundancy. The TYPE-component of the abstract substitution β is not present for reasons of readability.

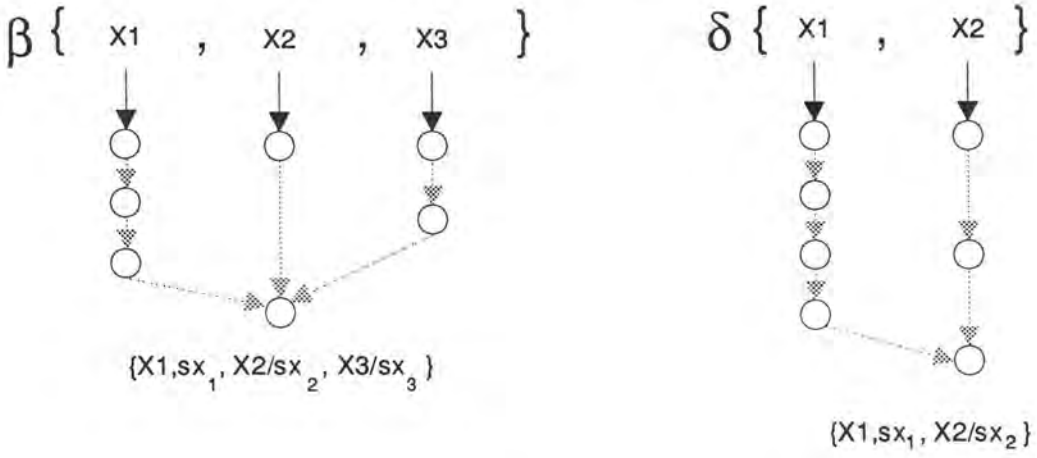


Figure 6.13 : *δ is subsumed by β*

A similar technique is used to found the presence of a subsumed component, we had to come down the tree representing the SVAL-component by the path defined in the new SVAL-constraint. We stop the descend at the node corresponding to the SVAL-node of SVAL(δ) (if it is possible to reach it, because if it isn't, there is in no possibility of a subsumed component). From this node, we continue the descend in the SVAL-constraint of SVAL(β) till the end. If we detect a node with a list of father which contains a subset of the variables from those present at the SVAL-node of SVAL(δ), we can be sure that this node is a subsumed component. This situation is illustrated by the next figure, where we can see that the path used the reach the SVAL-node in SVAL(δ) can be used to come down in SVAL(β). As in the previous figure, the TYPE-component of the abstract substitution β is not present for reasons of readability.

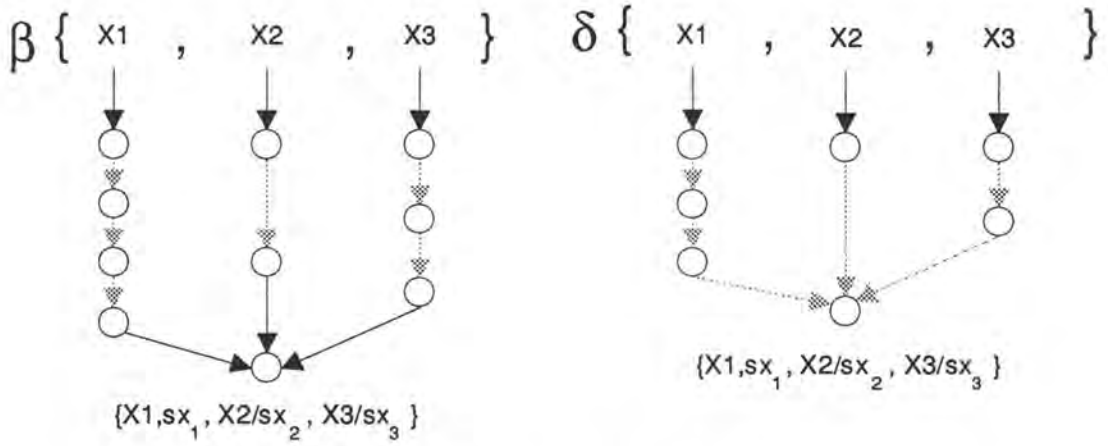


Figure 6.14 : *β is subsumed by δ*

Once the subsumed component detected, if suffice to destroy the branch of the tree which lead to this merged node to remove it.

After the addition of the new SVAL-constraint by Add_Constraint_On_Existing_Subst, the abstract substitution could be not normalized because the later procedure doesn't modify the type-graph and thus it could be a problem of compatibility between the type and the same-value. So we have to re-force this compatibility , this is the job of the procedure StandardizeSubst.

6.4.3. Formalization of the algorithm

6.4.3.1. Function detecting if SVAL(δ) is subsmed by SVAL(β) :

This function receive two abstract substitutions δ and β and returns TRUE if the abstract substitution β contains a SVAL-constraint which subsume the SVAL-constraint in SVAL(δ). We express the fact that :

- the SVAL-constraint in SVAL(δ) would be placed deeper in the tree representing the SVAL-component in β than an corresponding SVAL-constraint of β,
- all the variables involved in δ are involved by the existing SVAL-constraint of SVAL(β) too.

So, the introduction of the SVAL-constraint of SVAL(δ) will be redundant with the existing one, so it is not necessary to include it in the SVAL-component, and we decide to throw it away.

More formally, we can write :

Is_Subsumed_By(δ,β) : this function return TRUE ⇔

$$\forall \left\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\right\} \in SVAL(\beta)\ ,\ \neg\exists \left\{Y_1/S_{Y_1},\ldots,Y_m/S_{Y_m}\right\} \in SVAL(\delta):$$
$$\forall\ Y_j \in \delta, \exists X_i \in \beta : Y_{S_j} = X_{S_i}.s \quad (n \geq m)$$

6.4.3.2. Function detecting if SVAL(δ) subsme SVAL(β) :

This function receive two abstract substitutions δ and β and returns TRUE if the abstract substitution β contains a SVAL-constraint which is subsumed by the SVAL-constraint in SVAL(δ). We express the fact that :

- the SVAL-constraint in SVAL(δ) would be placed higher in the tree representing the SVAL-component than an corresponding SVAL-constraint of β,
- all the variables involved in the existing SVAL-constraint of β are involved by SVAL(δ).

So, the SVAL-constraint (subsumed by the new one) which is already present in the abstract substitution β must be removed from the abstract substitution because it will be redundant after the addition of the new constraint expressed in SVAL(δ).

More formally, we can write :

Subsume(δ,β) : this function return TRUE ⇔

$$\forall\left\{Y_1/S_{Y_1},\ldots,Y_m/S_{Y_m}\right\} \in SVAL(\delta)\ ,\neg\exists \left\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\right\} \in SVAL(\beta):$$
$$\forall X_i \in \beta : \exists\ Y_j \in \delta : X_{S_i} = Y_{S_j}.s \quad (m \geq n)$$

6.4.3.3. Function removing the subsumed constraint :

This function remove all the SVAL-constraints on the abstract substitution which realise the following condition :

$$\forall\left\{Y_1/S_{Y_1},\ldots,Y_m/S_{Y_m}\right\} \in SVAL(\delta)\ ,\exists \left\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\right\} \in SVAL(\beta):$$
$$\forall X_i \in \beta : \exists\ Y_j \in \delta : X_{S_i} = Y_{S_j}.s \quad (m \geq n)$$

6.4.3.4. Procedure adding a SVAL-constraint :

This procedure permit to add a new SVAL-constraint on an abstract substitution β without any concern about the compatibility between the type-graph and the new SVAL-constraint. So after the execution of this procedure, the abstract substitution could be non normalized.

β_{out} = Add_Constraint_On_Existing_Subst(δ,β_{in}) ;

PreCondition :

- β_{in} is a normalized abstract substitution, $\gamma(\beta_{in})$ is the concretization of β_{in}
- (δ) is a SVAL-constraint which is not subsumed by a SVAL-constraints already presents in SVAL(β).

PostCondition :

- β_{in} and thus $\gamma(\beta_{in})$ have not changed
- β_{out} is a normalized abstract substitution
- $\gamma(\beta_{out})$ the concretization of β_{out} is the same $\gamma(\beta_{in})$ except that there is a new SVAL-constraint on SVAL(β_{out}) which was present in SVAL(δ).

6.4.3.5. Procedure of standardization :

This procedure permit to enforce the compatibility between the type-graph and the same-value in the sense that when a set of variables is concern with a SVAL-constraint that means that they have the same subtype's denotation.

β_{out} = StandardizeSubst(β_{in}) ;

PreCondition :

- β_{in} is any abstract substitution.

PostCondition :

- β_{out} is such that : $\forall\left\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\right\} \in SVAL(\beta_{out}) : n_0^{X_1}/S_{X_1} \equiv \ldots \equiv n_0^{X_n}/S_{X_n}$

As the reader can observe, the algorithm of normalization implemented in this work is far from those proposed in the thesis.

6.4.4. A new specification of the algorithm Normalize

β_{out} = Normalize(β_{in},δ) ;

PreCondition :

- β_{in} is a normalized abstract substitution
- is an abstract substitution representing an unique SVAL-constraint of the form $\{X_1/s_{X_1},\ldots,X_n/s_{X_n}\}$ which must be added on the abstract substitution β_{in}.

PostCondition :

$\forall\theta : \theta \in \gamma(sval(\beta) \cup sval(\delta)) \Leftrightarrow \theta \in (\beta_{out})$

β_{out} is a normalized $\Leftrightarrow$

% it exists no loop in SVAL(β_{out}) %

$\forall\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\} \in SVAL(\beta_{out}) : (X_i/S_{X_i}, X_i/S_{X_i}.s \in SVAL(\beta_{out}) \Rightarrow S_{X_i} \neq S_{X_i}.s \ \forall s$

% type graphs and same value are compatible %

$\forall\{X_1/S_{X_1},\ldots,X_n/S_{X_n}\} \in SVAL(\beta_{out}) : n_0^{X_1}/S_{X_1} \equiv \ldots \equiv n_0^{X_n}/S_{X_n}$

% it exists no top-down subsumption %

$\forall S = \{X_r/S_{X_r},\ldots,X_s/S_{X_s}\} \in SVAL(\beta_{out}) \ \exists T = \{X_p/S_{X_p},\ldots,X_q/S_{X_q}\} \in SVAL(\beta_{out}) :$

such as $\{X_p,\ldots,X_q\} \subseteq \{X_r,\ldots,X_s\}$ and $\exists s : \forall X_i \in T, \exists X_j \in S$ so that $X_i = X_j.s$

% it exists no bottom-up subsumption %

$\forall S = \{X_r/S_{X_r},\ldots,X_s/S_{X_s}\} \in SVAL(\beta_{out}) \ \exists T = \{X_p/S_{X_p},\ldots,X_q/S_{X_q}\} \in SVAL(\beta_{out}) :$

such as $\{X_p,\ldots,X_q\} \subseteq \{X_r,\ldots,X_s\}$ and $\exists s : \forall X_i \in S, \exists X_j \in T$ so that $X_i = X_j.s$

6.5. *Several algorithms on the abstract substitutions*

In this section, we shall make an overview on the main difficulties appearing in the definition inspiring algorithms which manipulate the abstract substitutions.

The problems of transforming the definition proposed into algorithm is always the same, a lot of difficulties are hidden behind the formalism used and reveal there presence at the implementation time. An example of this situation is expressed by the following definition which inspired of the definition created to compute the upper bound of two abstract substitutions.

$\forall \beta_1, \beta_2 \in A_N$: $upp(\beta_1, \beta_2)$ =

$$\text{Type}(\delta) = \left\{ X_i \leftarrow \text{restrict}(n_{X_i}) \;\middle|\; \begin{array}{l} X \in D \text{ and} \\ \text{is an OR-node with two outgoing forward arcs:} \\ \text{one to the root of } T^{\beta'}_{X_i} \text{ and on to the root of } T^{\beta^2}_{X_i} \end{array} \right\}$$

$$\text{SVAL}(\delta) = \left\{ \{X_1 / S_{X_1}, \ldots, X_n / S_{X_n}\} \;\middle|\; \begin{array}{l} \beta_p, \beta_s \in \{\beta_1, \beta_2\} \text{ and } \beta_p \neq \beta_s \text{ and} \\ \exists \{X_1 / S_{X_1}, \ldots, X_n / S_{X_n}\} \in SVAL(\beta_s) \exists \{X_1 / P_{X_1}, \ldots, X_n / P_{X_n}\} \in SVAL(\beta_s) \\ \exists\, s : S_{X_1} = P_{X_1}.s, \ldots, S_{X_n} = P_{X_n}.s \text{ and} \\ S_{X_1}, \ldots, S_{X_n} \text{ are determinate selector in } T^{\beta}_{X_1}, \ldots, T^{\beta}_{X_n} \end{array} \right\}$$

This definition implies several problems. Once the SVAL-node of an abstract substitution is localised, we must found the corresponding one into the other abstract substitution. To realise this operation, we proceed like in the detection of the subset i.e. we come down by the selector associated with a variable of the first abstract substitution into the type-graph associated with the second abstract substitution. Once both SVAL-node are found, we must add the higher node of the tree (and thus the shortest selector) into the new abstract substitution.

This operation is delicate because the computation of the type-graph resulting of those presents in can done a completely different type-graph. The following example illustrate this situation.

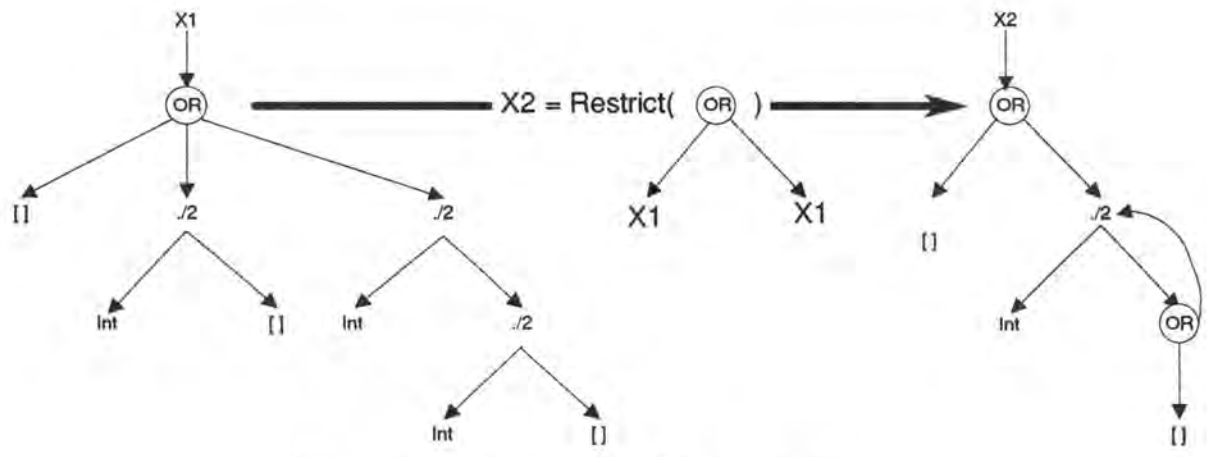


***Figure 6.15** : An example of the effect of Restrict*

As explained by the previous figure, the type-graph resulting of the restrict operation could be very different than the type-graph which served to its creation. So we had to construct an operation which add the SVAL-constraint only where they are valid.

The same problem arise when we must add all the SVAL-constraints present in twice abstract substitution on a new one. This situation can be found during the operation called BackWard Unification.

$$\forall \beta_i, \beta_r \in A_N\text{: bunif}(\beta_i, \beta_r) = \text{if } (\beta_i = \bot \text{ or } \beta_r = \bot) \text{ then } \bot \text{ else normalize}(\beta_n) \text{ with}$$

$$\text{Type}(\beta_n) = \left\{ X_j \leftarrow \text{btunif}(T^{\beta_i}_{X_j}, T^{\beta_r}_{X_j}) \mid X \in D \right\}$$

$$SVAL(\beta_n) = SVAL(\beta_r) \bigcup SVAL(\beta_i)$$

Hopefully, we have already build the algorithm will resolve those problems, it is Add_constraint. An interested reader can verify that the specification of this later algorithm meet perfectly the requirement of the question raised above.

6.6. *Problems derived from the data structures*

The data structures presented before has many advantages but they suffer from some inconvenient too.

- The first major problem is to copy an abstract substitution, indeed, when we copy a SVAL-constraint, we had to build the list of father but it could happen that the father has not been created yet.
- The principle of merging the tree forming the SVAL-component imposes to be very prudent when desallocating a node. For example, if we decide to destroy a node which was still used by a SVAL-constraint, the allocation error is certain for the next acces on the abstract substitution containing the later SVAL-constraint.

Now that we have make an overview of the principal problems derived from the data structures we have to examine solutions founded to resolve it. This can be founded in the next section.

6.6.1. Copying an abstract substitution

As explained before, the main difficulties in copying an abstract substitution is that we have to make reference to objects which have not been created yet. For example, let us suppose that we are copying the first variable of the abstract substitution β, and let us suppose that this variable is linked by an SVAL-constraint on the variable X_2. By means of our data structure, those two variables share a node called the SVAL-node from which starts a list of father containing as many cells as the number of variables sharing the node. In those cells, we can find a reference to the father of this node, and so to the variable X_2, the problem is that the father can't exist because we haven't copied the variable X_2 yet, as shown in the next figure.

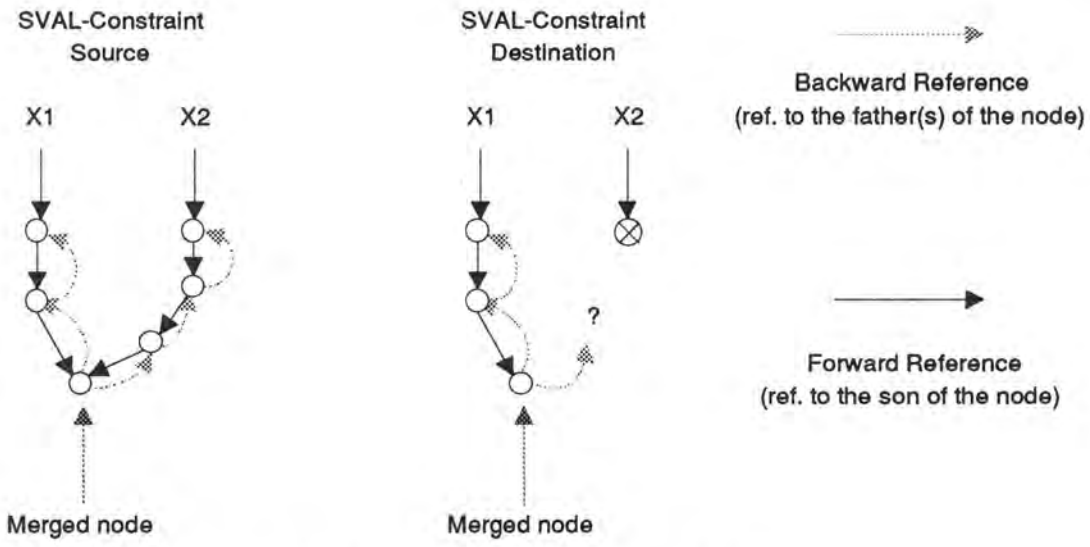


***Figure 6.16** : Example of copying a substitution*

The solution found to realize cleanly this operation is to create two recursive procedures, the first one builds the SVAL-component in a top-down way whereas the other ones builds the SVAL-component in a bottom-up way.

$\beta_{dest.}$ = CopySvalComponent(β_{source}) ;

PreCondition :

- β_{source} is normalized abstract substitution.

PostCondition :

- $\beta_{dest.}$ is normalized abstract substitution copy of β_{source}.
- β_{source} is normalized abstract substitution.

Structure of the algorithm :

```
Copy_Top_Down(node )
↓
nd_res = Copy_Node(node);
While (Exist_Son_To_be_copied(node))
    { Copy_Top_Down(node → son_i) }
While (Exist_Father_To_be_Copied(node))
    {if (Unknown_Reference(node → father_i))
       then Copy_Bottom_Up(node → son_i )
       else Copy_Father(node → father_i)}
Attach_List_of_Father(nd_res);
```

```
Copy_Bottom_Up(node )
↓
nd_res = Copy_Node(node);
While (Exist_Father_To_be_copied(node))
   {if (Unknown_Reference(node → father_i))
      then Copy_Bottom_Up(node → Father_i )
      else Copy_Father(node → father_i)}
While (Exist_Son_To_be_Copied(node))
    {if (Son_Already_Copied(node → son_i))
       then Insert_In_List_Son(node → son_i)
       else Copy_Top_Down(node → son_i)}
Attach_List_of_Son(nd_res);
```

This algorithm needs following comments :

- the left-hand part of this algorithm is the one which permits to create all the sons of a node and thus is dedicated to build the SVAL-constraint in a Top-Down way. When it detects a reference to an object which has not been created yet, it calls the right-hand part and it puts the result given by the call into the place where the unknown reference was done.
- The right-hand part of the algorithm allows to create an SVAL-constraint in an Bottom-Up way, it is thus dedicated to create the list of fathers of a node as the reference to the father it-self. Once it has build the upper part of the SVAL-constraint, it calls the left-hand size part and it puts the result given by the call as the list of son of the node he has created.

This algorithm is a bit complex but it must have good performance because it will be called often, so the idea to copy an SVAL-component in one pass was necessary to speed up this time-consuming operation.

6.6.2. Deletion of an abstract substitution

The major problem of this operation is to know when a node can be freed, this is not as simple as it seems to be because a node can be shared by several others. The situation will be more viable if we know before killing a node how much time it is referenced. This is the job of the field *NbSharing* in the node denoting a functor. The algorithm of deletion becomes very simple with a good utilisation of this information. We can see in the next figure, how to deal when we wanted to delete the abstract substitution β composed by three variables on which there is an unique SVAL-constraint forming the SVAL-component (the type-graph's are not present in this figure for obvious reasons of readability) .

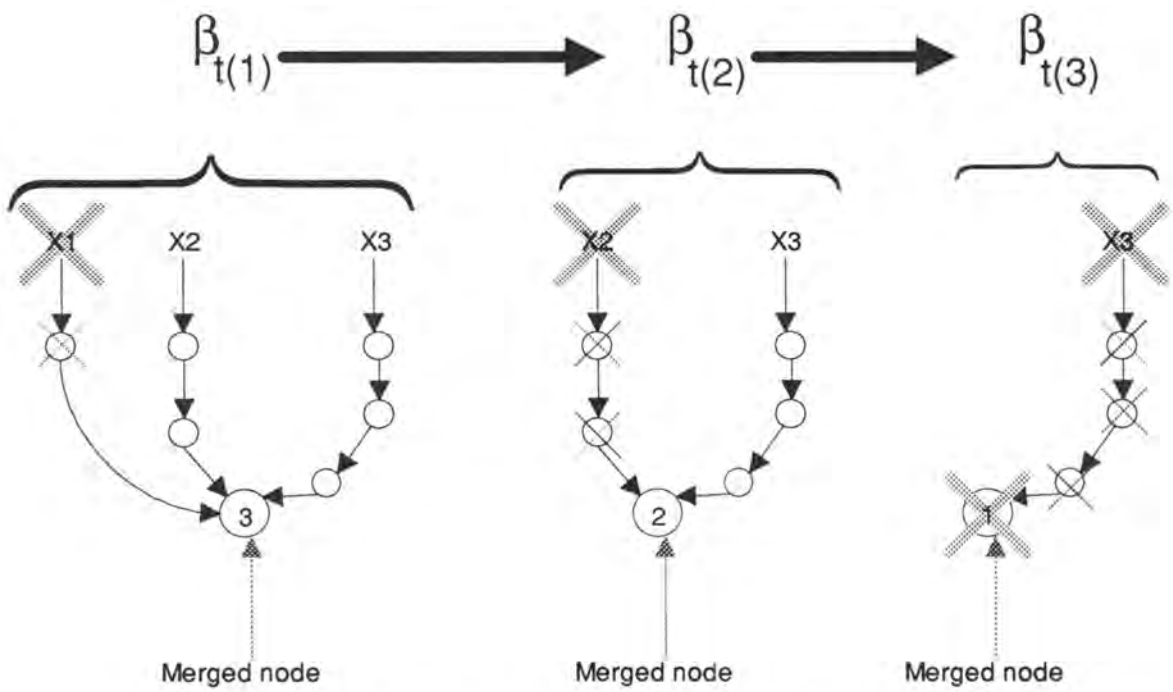


***Figure 6.17** : Example of deletion of a substitution*

Free_Sval_Constraint(SVAL-node) ;

PreCondition :

- $(\text{SVAL}-\text{node} \rightarrow NbSharing_0) >= 1$
- SVAL-node is a valid functor node

PostCondition :

- SVAL-node is still valid $\Leftrightarrow (\text{SVAL}-\text{node} \rightarrow NbSharing_0) > 1$

As valid, we means that the SVAL-node has not been desallocated of the memory.

6.7. *Summary*

In this chapter, we have presented the data structures chosen to represent the SVAL-component as the abstract substitution (Section 6.2 and 6.3). After a discussion on the changes bring out by the proposed representation, we were focusing the algorithm Normalize. We have proposed a new version of this algorithm which performs the normalization of the abstract substitution immediately during the addition of the SVAL-constraint.

A sparing description of the problems (as of the solutions used) posed by the algorithm working on the abstract substitutions is done in the section 6.5.

The latest section is dedicated to the tricks bring out by the chosen data structure.

Chapter 7

Abstract interpretation operations

7.1. Introduction

All the operations defined previously were designed to be integrated on the abstract interpreter developped by [2]. The first section of this chapter describes the abstract operations permitting to interface with this abstract interpreter. Those operations are proposed to the attention of the reader to be complete, they were not implemented in this work because another abstract interpreter was used.

Second section describes the operations which are necessary to the abstract interpreter used and which was developped by [8]. This algorithm is not described here because it is not the purpose of this work. We just describe informally all the operations which form the interface between the domain and the abstract interpreter. An attentive reader can find in the following papers a precise specification of the abstract operations in terms of the concretization functions [8],[9],[10].

7.2. Original abstract interpreter for that abstract domain

By original, we means that this abstract domain was conceived for this abstract interpreter and not for the abstract interpreter used. This section of the chapter is strongly inspired by [7], the reader can refer to this work for more information.

7.2.1. Procedure-entry(P,β_{in})

Assume that $\{Y_1,...,Y_n,X_1,...,X_p\}$ is the domain of the call-substitution β_{in} and that the call is $P(Y_1,...,Y_n)$ for which a set of clauses $P(Z_1^j,...,Z_n^j) \leftarrow B_1^j,...,B_{j_n}^j$ exitsts. Assume that $\{Z_1^j,...,Z_n^j,Z_{n+1}^j,...,Z_q^j\}$ are the variable of the j^{th} clause. Procedure entry has to compute β_{in}^r and β_j^{in}.

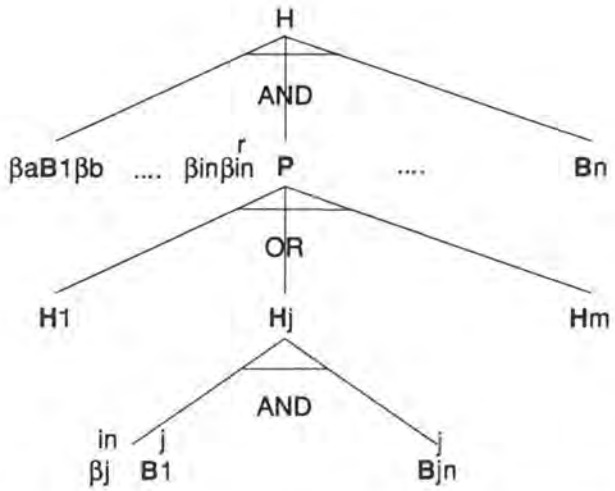


***Figure 7.1** : Procedure-entry extends the abstract AND-OR graph*

Definition 7.1 :

$$TYPE\left(\beta_{in}^{r}\right)=\left\{W \leftarrow T_{w}^{\beta_{in}} \middle| W \in \{Y_1,....,Y_n\}\right\}$$

$$SVAL\left(\beta_{in}^{r}\right)=\left\{\{W_1/s_1, W_2/s_2\} \in SVAL\left(\beta_{in}\right) \middle| W_1, W_2 \in \{Y_1,....,Y_n\}\right\}$$

$$\beta_{in}^{r}=R\left(\beta_{in}^{r}\right)$$

$$TYPE\left(\beta_{j}^{in}\right)=\left\{Z_1^j \leftarrow T_{y_1}^{\beta_{in}^{r}},...,Z_n^j \leftarrow T_{y_n}^{\beta_{in}^{r}}, Z_{n+1}^j \leftarrow \max,...,Z_q^j \leftarrow \max\right\}$$

$$SVAL\left(\beta_{j}^{in}\right)=\left\{\{Z_k^j/s_k, Z_l^j/s_l\} \middle| \{Y_k/s_k, Y_l/s_l\} \in SVAL\left(\beta_{in}^{r}\right)\right\}$$

Note that we do not deal explicitely with the trivial case β_{in} of being $\perp$ as the result of any abstract operation on is $\perp$ again $\perp$.

7.2.2. Procedure-exit(P,β_{in},{...,β_j^{out},...})

The same assumptions as Procedure-entry are made. Let β_j^{out} be the abstract sucess-substitution of $\beta_{j_n}^{j}$. Procedure-exit has to compute β_{out}^{r} and β_{out} (see figure 7.2). β_{out}^{r} over the domain $\{Y_1,...,Y_n\}$ is defined as follows :

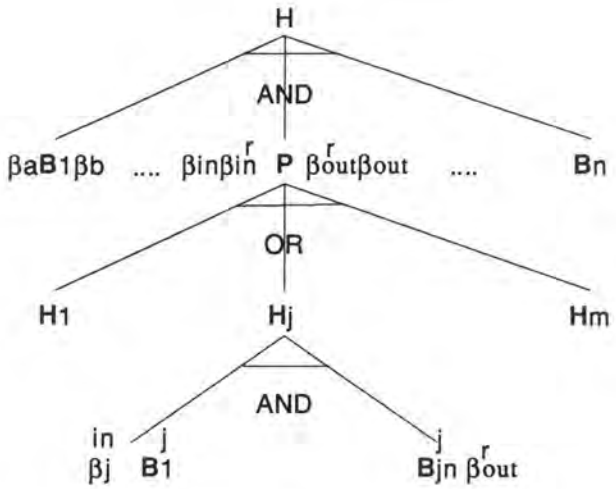


Figure 7.2 : *Procedure-exit computes* βout

Definition 7.2 :

$$TYPE\left(\beta^{'j}\right)=\left\{Y_1 \leftarrow T_{Z_1^j}^{\beta_j^{out}},\ldots,Y_n \leftarrow T_{Z_n^j}^{\beta_j^{out}}\right\}$$

$$SVAL\left(\beta^{'j}\right)=\left\{\left\{Y_k / s_k, Y_l / s_l\right\} \middle| \left\{Z_k^j / s_k, Z_l^j / s_l\right\} \in SVAL\left(\beta_j^{out}\right) \text{ and k, } 1 \in [1,n]\right\}$$

$\beta' = upp(\beta'^1, upp(\ldots, upp(\beta'^{p-1}, \beta'^p)\ldots))$ with p the number of defining clauses.

$\beta_{out}^r = bunif(\beta_{in}^r, \beta')$

Computing β_{out} from β_{out}^r and β_{in} (*the extansion operation*) has to take into account the dependencies between X_q and Y_r. The possible values for such a variable X_q which did not participate directly in the call can become more restricted due to an existing dependency with a Y_r.

Definition 7.3 :

$$TYPE\left(\beta_{out}\right)=\left\{\ldots,Y_j \leftarrow T_{Y_j}^{\beta_{out}^r},\ldots,X_n \leftarrow normalize(T_{X_i}^{'}),\ldots\right\}$$

with $T_{X_i}^{'}$ =

if $\exists \left\{X_i / s_i, Y_j / s_j\right\} \in SVAL\ (\beta_{in})$:

(not ($\exists \left\{X_i / s_i^{'}, Y_k / s_k\right\} \in SVAL(\beta_{in})$: s_i extends $s_i^{'}$ and $s_i^{'} \neq s_i$)

and not ($n_0^{X_i\beta_{in}} / s_i \equiv n_0^{Y_j\beta_{out}^r} / s_j$))

replace($T_{X_i}^{\beta_{in}}$, btunif ($n_0^{X_i\beta_{in}} / s_i, n_0^{Y_j\beta_{out}^r} / s_j$))

else $T_{X_i}^{\beta_{in}}$

$SVAL(\beta_{out}) = SVAL(\beta_{in}) \cup SVAL(\beta^r_{out})$

β_{out} = normalize(β_{out})

During the extension step we look for a subgraph T_{X_i} / s_i that describes in β_{in} terms that have to be identical to subterms of a Y_j, namely in the denotation of T_{X_j} / s_j. We do not have to consider any subgraphs of such a T_{X_i} / s_i, because the type-graphs in β^r_{out} are normalized and thus compatibility between types and SVAL-constraints is assured.

For such a type graph we check whether the denotations are still identical after the call. If not, we change the denotation of T_{X_i} / s_i by btunif $(n_0^{X_i\beta_{in}} / s_i, n_0^{Y_j\beta^r_{out}} / s_j)$ which retains only those bindings that are possible instantiations of the terms in $ID(n_0^{X_i\beta_{in}} / s_i)$.

The values of X_i may become instantiated by the call due to sharing of a free variable with a Y_j. In this case, we know that the sharing occurs in a subterm t_S such that t_S and $t_S\theta$ belong to the denotation of a max-node. So no special action must be performed during the extension to deal with this sharing. The SVAL-constraints in β_{out} have determinate selectors.

This is trivial for those in SVAL(β^r_{out}). Those from SVAL(β_{in}) are determinate due to the proposition 3.19 of btunif.

7.2.3. Abstract-interpretation-built-in(P,β_{in})

This operation is divided in three parts. The first part computes β^{ra}_{in} and corresponds to the first part of Procedure-entry but we do not apply the opertor R. The last part computes β_{out} from β_{in} and β^r_{out} and corresponds to the last part (extension) of Procedure-entry. So we only have to consider the middle step, computing β^r_{out} from β^{ra}_{in}.

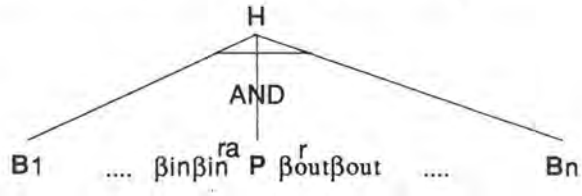


***Figure 7.3** : Abstract interpretation of the built-in P*

7.2.3.1. Abstract interpretation of X = Y

The intended denotation of the abstract success-substitution is the set

$$\left\{ t_x\sigma \middle| \left\{ X \leftarrow t_x, Y \leftarrow t_y \right\} \in \gamma(\beta_{in}^{ra}) \text{ and } \sigma \text{ is a mgu of } t_x \text{ and } t_y \right\}$$

This set is contained in the set

$$\left\{ t_x\sigma \middle| t_x \in \mathrm{ID}(\mathrm{T_x}) \text{ and } t_y \in \mathrm{ID}(\mathrm{T_y}) \text{ and } \sigma \text{ is a mgu of } t_x \text{ and } t_y \right\}$$

Proposition 7.1 :

$$\left\{ t_x\sigma \middle| t_x \in \mathrm{ID}(\mathrm{T_x}) \text{ and } t_y \in \mathrm{ID}(\mathrm{T_y}) \text{ and } \sigma \text{ is a mgu of } t_x \text{ and } t_y \right\} = \left\{ t \middle| t \in \mathrm{ID}(\mathrm{T_x}) \text{ and } t \in \mathrm{ID}(\mathrm{T_y}) \right\}$$

Note that the latter set on Proposition 7.1 is just the denotation computed by intersection(T_X,T_Y).

Definition 7.4 :

$$TYPE\left(\beta_{out}^{r}\right) = \left\{ X \leftarrow T_u, Y \leftarrow T_u \right\} \text{ with } \mathrm{T_u} = \text{intersection}\left(T_X^{\beta_{in}^{ra}}, T_Y^{\beta_{in}^{ra}}\right)$$

$$SVAL\left(\beta_{out}^{r}\right) = SVAL\left(\beta_{in}^{r}\right) \cup \left\{ \left\{ X/\varepsilon, Y/\varepsilon \right\} \right\}$$

$$\beta_{out}^{r} = \text{normalize}(\beta_{out}^{r})$$

Theorem 7.3 : Abstract interpretation of X=Y is correct.

7.2.3.2. Abstract interpretation of X = f(Y_1,...,Y_k)

Definition 7.5 :

$$TYPE\left(\beta_{out}^{r}\right) = \left\{ X \leftarrow T_u, \ldots, Y_i \leftarrow \text{restrict}\left(n_0^{X\beta_{out}^{r}} / (\mathrm{f/k,i})\right), \ldots \right\}$$

where T_u = intersection$\left(T_X^{\beta_{in}^{ra}}, T_Y\right)$ and T_Y a type graph with $\mathrm{lb}\left(n_0^{Y}\right) = \mathrm{f/k}$ and k sons such that $n_0^{Y}/i = T_{Y_i}^{\beta_{in}^{ra}}$.

$$SVAL\left(\beta_{out}^{r}\right) = SVAL\left(\beta_{in}^{ra}\right) \cup \left\{ \ldots, \left\{ X/(\mathrm{f/k},i), Y_i/\varepsilon \right\}, \ldots \right\}$$

$$\beta_{out}^{r} = \text{normalize}(\beta_{out}^{r})$$

Observe that for each Y_i we construct a new type-graph whose denotation is the same as the one of the i^{th} son of the root of T_u which is a functor node with label f/k.

Theorem 7.4 : Abstract interpretation of $X = f(Y_1,...,Y_k)$ is correct.

7.3. Abstract interpreter used

The operations presented below are conceived to deal with the various situations encountered during the computation. Two operations deal with the unification. There is an operation for each form the unification can have, either a unification of variables or the unification of a variable and a functor. They are two operations dedicated to the procedure clause. The first one is needed to extend the abstract substitution to the variables present in the bodies of the clause, the other one is used at the end of the work on the clause to restrict the number of variables at those presents in the head of the clause. The last pair of operations is related to the handling of the procedure call. The first operation restricts the number of variables at those present on the call. The second one realize the propagation of the result inside the substitution.

After the presentation of the operation conceived to deal with the various situations encountered during the computation, we describe an operation needed by the abstract interpreter and which allows to computes the least upper bound of two abstract substitutions.

7.3.1. Unformal description

AI-VAR(β) where β is an abstract substitution on $\{X_1,X_2\}$: this operation returns the abstract substitution obtained from β by unifying variables X_1,X_2. It is used for goals of the form $X_i = X_j$ in normalized programs.

AI-FUNC(β,*f*) where β is an abstract substitution on $\{X_1,...,X_n\}$ and *f* is a function symbol of arity n-1: this operation returns the abstract substitution obtained from β by unifying X_1 and $f(X_2,...,X_n)$. It is used for goals of the form $X_{i1} = f(X_{i2},...,X_{in})$ in normalized programs.

EXTC(c,β) where β is an abstract substitution on $\{X_1,...,X_n\}$ and *c* is a clause containing variable $\{X_1,...,X_m\}$ $(m \geq n)$: this operation returns the abstract substitution obtained by extending β to accomodate the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head. In logical terms, this operation, together with the next operation, achieves the role of the exitential quantifier.

RESTRC(*c*,β) where β is an abstract substitution on the clause variables $\{X_1,...,X_m\}$ and $\{X_1,...,X_n\}$ are the head variables of clause *c* $(n \leq m)$: this operation returns the ABS obtained by projecting β on variables $\{X_1,...,X_n\}$. It is used at the exit of a clause to restrict the substitution to the head variables only.

RESTRG(*g*,β) where β is an abstract substitution on $D = \{X_1,...,X_n\}$, and *g* is a goal $p(X_{i1},...,X_{im})$ (or $X_{i1}=X_{i2}$ or $X_{i1} = f(X_{i2},...,X_{im})$) : this operation returns the ABS obtained by

1. projecting β on $\{X_{i1},...,X_{im}\}$ obtaining β';

2. expressing β' in terms of $\{X_1,...,X_m\}$ by mapping X_{ik} or X_k.

It is used before the execution of a goal in the body of a clause. The resulting substitution is expressed in terms of $\{X_1,...,X_m\}$, i.e. in the same way as the input and output substitutions of p in the abstract domain.

EXTG(g,β, β') where β is an abstract substitution on $D = \{X_1,...,X_n\}$, the variables of the clause where g appears, g is a goal $p(X_{i1},...,X_{im})$ (or $X_{i1}=X_{i2}$ or $X_{i1} = f(X_{i2},...,X_{im})$) with $(X_{i1},...,X_{im}) \subseteq D$ and β' is an ABS on $\{X_1,...,X_m\}$ representing the result of $p(X_1,...,X_n)\beta''$ where $\beta'' = RESTRG(g,\beta)$: this operation returns the ABS obtained by extending β to take into account the result β' of the goal g. It is used after the execution of a goal to propagate the results of the goal on the substitution for all the variables of the clause.

UNION $\{\beta_1 ,...,\beta_n\}$ where $\{\beta_1 ,...,\beta_n\}$ are the abstract substitutions from the same cpo : this operation returns an abstract substitution representing all the substitutions satisfying at least one β_i. It is used to compute the output of a procedure given the outputs for its clauses.

All these definitions are extracted form [8] and can be illustrated by the following figure.

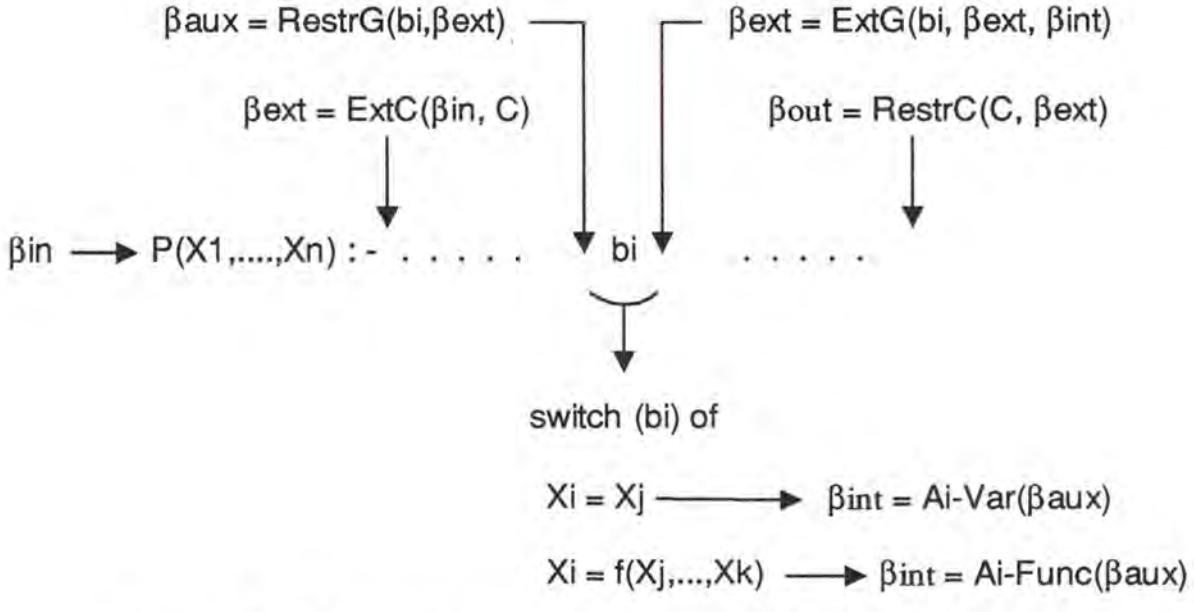


***Figure 7.4** : Abstract interpretation operation for the interpreter used*

7.3.2. Adaptation on the defined domain

7.3.2.1. $\beta_{ext} = ExtC(\beta_{in}, C)$

Let $D = \{X_1,\ldots,X_n\}$ be the domain of β_{in} and $D' = \{X_1,\ldots,X_{n+k}\}$ be the set of variables in the clause C. $ExtC(\beta_{in},C)$ produces an abstract substitution β_{ext} such that:

$$TYPE(\beta_{ext}) = \{X_1 \leftarrow T_{X_1}^{\beta_{in}},\ldots,X_n \leftarrow T_{X_n}^{\beta_{in}}, X_{n+1} \leftarrow \max,\ldots,X_{n+k} \leftarrow \max\}$$

It is a simple mapping from the variables of β_{in} to the variables of β_{ext}. The variables added to the variables of β_{in} to satisfy the number of variables present in the clause C are associated with a max-node as type-graph.

$$SVAL(\beta_{ext}) = SVAL(\beta_{in})$$

The same-value of β_{in} is just mapped on β_{ext}. There is no same-value component on the new variables β_{ext}.

7.3.2.2. $\beta_{aux} = RestrG(b_i,\beta_{ext})$

Let $D' = \{X_1,\ldots,X_{n+k}\}$ be the domain of β_{ext} and $D'' = \{X_{i_1},\ldots,X_{i_q}\}$ be the set of variables in the body of a goal b_i $(D'' \subseteq D')$. $RestrG(b_i,\beta_{ext})$ produces a subst β_{ext} obtained by:

$$TYPE(\beta_{aux}) = \{X_1 \leftarrow T_{X_r}^{\beta_{ext}},\ldots,X_q \leftarrow T_{X_s}^{\beta_{ext}}\}\ \forall r,s: 1 \le r \ne s \le n+k$$
$$SVAL(\beta_{aux}) = \{\{X_n/S_n,\ldots,X_m/S_m\} \mid \{X_i/S_i,\ldots,X_j/S_j\} \in SVAL(\beta_{ext})\}\ \forall n,m: 1 \le n \ne m \le q$$

We only take into account the variables of β_{ext} which are used in the goal b_i and we transfer the value of the variables from β_{ext} to β_{aux}.

7.3.2.3. $\beta_{int} = Ai\text{-}Var(\beta_{aux})$

$\beta_{int} = Ai\text{-}Var(\beta_{aux})$ where β_{aux} is an abstract subst. on $\{X_1,X_2\}$.
This operation reurns the abstract subst. obtained from β_{aux} by unifying variables X_1, X_2. It is used for goals of the form $X_i = X_j$ in normalized programs.

$$TYPE(\beta_{int}) = \{X_1 \leftarrow T_u, X_2 \leftarrow T_u\}$$
with T_u = intersection$\left(T_{X_1}^{\beta aux}, T_{X_2}^{\beta aux}\right)$

$SVAL(\beta_{int}) = \{\{X_1/\varepsilon, X_2/\varepsilon\}\}$

Because of the implementation of Normalize, there is no need perform any operation of normalization on β_{int}. As matter of fact, the input abstract substitution β_{int} is normalized because we always keep the abstract substitution in a normalized form and the addition of $\{X_1/\varepsilon, X_2/\varepsilon\}$ is done by means of the procedure Add_constraint which keep the substitution normalized.

7.3.2.4. β_{int} = Ai-Func(β_{aux})

β_{int} = Ai-Func(β_{aux}) where β_{aux} is an abstract substitution on $\{X_1,...,X_q\}$ and f is a functional symbol of arity q-1. This operation returns the abstract substitution obtained from β_{aux} by unifying X_1 and $f(X_2,...,X_q)$. It is used for goals $X_{i_1} = f(X_{i_1},...,X_{i_e})$ in normalized programs.

$TYPE(\beta_{int}) = \{X_1 \leftarrow T_u,..., X_i \leftarrow \text{restrict}(n_0^{X\beta_{int}}/(f/k,i)),...\}$

where T_u = intersection$(T_{X_1}^{\beta_{aux}}, T_{X'})$ and $T_{X'}$ a type graph with $lb(n_0^{X'}) = f/k$ and k sons such that $n_0^{X'} = T_X^{\beta_{aux}}$ $(\forall i: 2 \le i \le q)$.

$SVAL(\beta_{int}) = SVAL(\beta_{int}) \cup \{\{X_1/(f/k,1), X_2/\varepsilon\},...,\{X_1/(f/k,k), X_k/\varepsilon\}\}$

As for Ai-Var, the same considerations can be done on the usefulness of performing Normalize.

7.3.2.5. β_{ext} = ExtG(bi, β_{ext}, β_{int})

Let $D'' = \{X_1,...,X_q\}$ be the domain of β_{int}. Let $D' = \{X_1,...,X_{n+k}\}$ be the domain of β_{ext} ($D'' \subseteq D'$). This operation returns the abstract substitution obtained by extending β_{ext} to take into account the result β_{int} of the goal b_i.

Let $f: D'' \rightarrow D'$ be a function, which from a variable X_i in input returns either X_j or $\varnothing$.

- If (ouput = X_j) then it means that X_i in D" is the same variable than X_j up to renaming.
- If (ouput = $\varnothing$) then it means that X_i has no corresponding variable in D'.

$TYPE(\beta_{ext}) = \{...,X_i \leftarrow T_{x_i}^{\beta},...\}$

with $T_{X_i}^{\beta} = if\ f(X_i) = X_j$

then restrict(replace($T_{X_i}^{\beta_{ext}}, s_i$, btunif($n_0^{X_i\beta_{ext}}/s_i, n_0^{X_i\beta_{int}}/s_i$)))

else $T_{X_i} = T_{X_j}^{\beta_{int}}$ $\forall j: 1 \le j \le q$

$$SVAL(\beta_{ext}) = \left\{ \left\{ f(X_i / s_i), \ldots, f(X_j / s_j) \right\} \middle| \left\{ X_i / s_i, \ldots, X_j / s_j \right\} \in SVAL(\beta_{int}) \right\}$$

All the operations performed during the extansion of Procedure-exit (Section 2) are done to establish compatibility between type-graphs and SVAL-constraints. All those operations are not neccessary here always for the same reason. We knows that β_{ext} is normalized because we always keep the abstract substitution in a normalized form. So, as we are using the procedure Add_constraint to add the all the SVAL-constraints contained in β_{int}, we obtain as result a normalized abstract β_{ext}.

7.3.2.6. β_{out} = RestrC(C,β_{ext})

Let $D = \{X_1, \ldots, X_n\}$ be the set of variables of β_{out} and $D' = \{X_1, \ldots, X_{n+k}\}$ be the set of variables of β_{ext} which are present in the clause C. This operation returns the abstract substitution obtained by projecting β_{ext} on the variables $\{X_1, \ldots, X_n\}$.

$$TYPE(\beta_{out}) = \left\{ X_1 \leftarrow T^{\beta_{ext}}_{X_1}, \ldots, X_n \leftarrow T^{\beta_{ext}}_{X_n} \right\}$$

$$SVAL(\beta_{out}) = \left\{ \left\{ X_i / S_i, \ldots, X_j / S_j \right\} \middle| \left\{ X_i / S_i, \ldots, X_j / S_j \right\} \in SVAL(\beta_{ext}) \right\} \ \forall i, j : 1 \le i \le j \le n$$

7.3.2.7. β_{out} = Union(β_{out},β_{ext})

The union of two abstract substitutions is the same operation than the upper-bound operation.

$\forall\ \beta_1, \beta_2 \in A_N$:
upp(β_1, β_2) =
if $\beta_1 = \perp$ then β_2
else if $\beta_2 = \perp$ then β_1
else δ with

TYPE(δ) =
{X $\leftarrow$ normalize(n_X) | X $\in$ D and
n_X is an OR-node with two outgoing forward arcs:
one of the root of $T^{\beta 1}_x$ and one to the root of $T^{\beta 2}_x$}
SVAL(δ) = {{X/s_X, Y/s_Y} |
$\beta_P, \beta_S \in \{\beta_1, \beta_2\}$ and $\beta_P \neq \beta_S$ and
$\exists$ {X/s_X, Y/s_Y} $\in$ SVAL(β_S) $\exists$ {X/p_X, Y/p_Y} $\in$ SVAL(β_P) $\exists$ s :
$s_X = p_X$. s and $s_Y = p_Y$. s and
s_X respectively s_Y are determinate selectors in T^{δ}_x respectively T^{δ}_y }

7.4. Summary

In this chapter, we present the operations interfacing the abstract interpreters. We present in detail the operations implemented. We discuss the changes bring from the implementation of Normalize on this operation.

Chapter 8

Interpretation of the results

The first results given by the abstract interpreter on this abstract domain where obtained in the same time than the redaction of this work. It was difficult for us to include those results in this paper because of the deadline. Furthermore, there were still some bugs in the program at this time, so, we have decided to add the results of the implementation in an addendum.

Chapter 9

Future optimization and conclusion

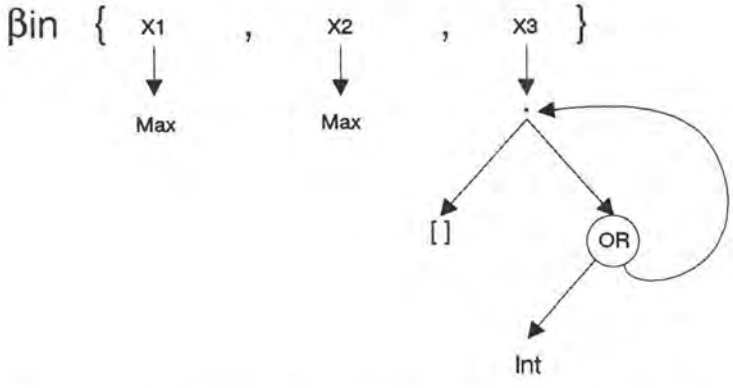
9.1. Optimization by memoization techniques

This technique is based on the following observation. A lot of operations on the type-graph are performed more than once with the same input, and thus gives the same output.

Example :
Let *append* be a normalized Prolog program.

```
append(X1,X2,X3) :-
        X1 = [],
        X3 = X2.
append(X1,X2,X3) :-
        X1 = [X4 | X5],
        X3 = [X4 | X6],
        append(X5,X2,X6).
```

Let be β_{in} the input of the program (there is no same value).



During the abstract interpretation of the Prolog Program with β_{in} the operation Compact is done more than 30 times on variable unified to List (see Appendix A for more).

A solution to avoid this situation is to create an array which memorizes for each existing type of operation its input and its output. This optimization, called *caching*, has already been performed on another domain, the reader can found its results in [18]. Once an operation is performed on the type of a variable, we parse the array to search if the willing operation has been done previously, if it is the case, there is no need to perform this operation because we already have its output, if it is not he case, we perform the operation and we store the output into the array.

Operation Compact

Input	Output
T0::=or(nil,.(int,t0))	T0::=or(nil,.(int,t0))
T0::=or(int,bottom,t0)	T0::=int
...	...

9.1.1. Discussion on the proposed optimization

The basic principle of this optimization is quiet simple but it is no certain that its implementation will improve the abstract interpretation.

Advantages :

This optimization is very useful on large trees, as matter of fact, performing an operation like Restrict on big trees is very time consuming. The computation time of the operation Restrict is replaced by the time required to copy the restricted tree from output of the array. It can be easily seen that copying a tree is faster than restricting a tree.

Drawnbacks :

Searching if an operation has already been performed implies to select the array corresponding to the desired operation and to compare each input of the array with the input of the current operation. It could take a lot of time, specially when the searched tree is not present in the array. In this case, the global computation time is very large because we must add the time required for the operation Restrict to the time of research.

Feasibility :

As expressed on Chapter 4, the algorithm Compact yields a normal form for the type-graphs. However, it is not an unique form. This can cause problems, as matter of fact, we have to parse the array to search, for an operation, about an input which is the same than the current type-graph. As the representation is not unique, we must apply the algorithm Equal on all the input of the array. Performing like this cost obviously a lot of time because of the application of the algorithm.

To have an efficient caching optimization, it is nessessary to possess a canonical form for the type graph, but it is not very easy to define it, mainly because of the OR-node. As matter of fact, the sons of the OR-node are not classified in a strict order, so to establish a canonical form, we must create a lexico-graphical order on the sons of the OR-node of the

type. With the help of this order, we can obtain an unique representation of the type-graph. So, we can colour each node of the type graph, and in summing all the coloration of the node, we can obtain a number which identify univoquely the tree, so there is no more need to use the operation Equal and the caching will begin to be efficient.

9.2. Conclusion

Abstract interpretation, as we already said, is an important tool to analyze statically Prolog programs and to improve performances of compilers.

A lot of work has been performed during this year to achieve the implementation of this abstract domain. It was a long task and the program corresponding to the domain is impressive (more that 12.000 lines of C only for the abstract domain). The first results were obtained at the end of the academic year. The abstract interpretor contains some 6.000 lines of C in the used version, which is the original version.

The first evaluation of the time requested by the abstract interpretation is quiet high, this is due to the strong complexity of the domain. However, it is not possible at the time to draw some conclusion about the efficiency of the implementation because of the reasons explained in Chapter 8.

Bibliography

[1] S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horkwood Limited, West Sussex, England, 1987.

[2] M. Bruynooghe, *A practical framework for the Abstract Interpretation of Logic Programs*, Journal of Logic Programming, 10(2):91-124, February 1991.

[3] M. Bruynooghe, G. Janssens, A. Marien and A. Mulkers, *The impact of abstract Interpretation: an experiment in code generation.*

[4] P. Cousot and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in Conf. Records of Fourth ACM Symposium on POPL, pp. 238-252, Los Angeles, CA, 1977.

[5] K. De Bosschere and L. Wulteputte, *Prolog Implementation Methods*, Technical report, Laboratorium voor Elektronica en Meettetechniek, 1992.

[6] V. Englebert, D. Roland, *Abstract Interpretation of Prolog Programs: Optimization of an Implementation*, Institute of Computer Science,University of Namur, Belgium, 1992.

[7] G. Janssens, *Deriving Run Time Properties of Logic Programs by Means of Abstract Interpretation*, PhD thesis, Katholiek Universiteit Leuven, Belgium, March 1990.

[8] B. Le Charlier and P. van Hentenryck, *Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog*, Fourth IEEE International Conference on Computer Languages (ICCL'92), San Francisco, CA, April 1992.

[9] B. Le Charlier, K. Musumbu and P. van Hentenryck, *Efficient and Accurate Algorithms for the Absract Interpretation of Prolog Programs*, Technical report 37/90, Institute of Computer Science, University of Namur, Belgium, 1990.

[10] B. Le Charlier, K. Musumbu, and P. van Hentenryck, *A Generic Abstract Interpretation Algorithm and its Complexity Analysis*, in K. Furukawa, editor, Proceedings of the Eighth International Conference on Logic Programming (ICLP'91), Paris, France, June 1991. MIT Press.

[11] B. Le Charlier, *L'Analyse Statique des Programmes par l'Interpretation Abstraite*, Institute of Computer Science, University of Namur, Belgium, 1992.

[12] B. Le Charlier, and P. van Hentenryck, *Reexecution in Abstract Interpretation of Prolog*, Proceedings of the International Joint Conference and Symposium on Logic Programming (JICSLP-92), Washington, DC, November 1992.

[13] C. Livercy, *Théorie des Programmes*, pp. 18-23, Dunod, 1978.

[14] J.W. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, New-York, 1984.

[15] K. Musumbu, *Interpretation Abstraite de Programmes Prolog*, PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990.

[16] P.J. Plauger, *The Standard C Library*, Prentice Hall, 1990.

[17] V. Englebert, B. Le Charlier, D. Roland, and and P. van Hentenryck, *Generic Abstract Interpretation Algorithm for Prolog : Two Optimization Techniques and their Experimental Evaluation*, Software Practice and Experience, 23(4), April 1993.

Appendix A

Note : this trace was obtained with the pre-version of the algorithm, it could be some bugs again.

```
append(X1 , X2 , X3 ) :- [ 0 ]
   X1 = [],
   X3 = X2.
append(X1 , X2 , X3 ) :- [ 3 ]
   X1 = [ X4 | X5 ] ,
   X3 = [ X4 | X6 ] ,
   append( X5 , X2 , X6 ).

AP- append(v,v,g)
OK ? n
verbose (y/n)? y
functor : append
arity : 3
T 0 ::= max
T 1 ::= end
T 0 ::= max
T 1 ::= end
T 0 ::= or([],.(int,t0))
T 1 ::= end

Entrez le nombre de composantes
formant cette partie de la same value <0,1=FIN> <2..9>: 0

TRY CLAUSE 1
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{ }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{ }

  EXIT UNIF-FUN
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{ }
```

```
  CALL UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT LUB
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 1
TRY CLAUSE 2
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
```

```
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  CALL UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(INT,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= .(INT,OR(T0,[]))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X3/(2)},{X6/(0)}  }

  CALL PRO-GOAL  append
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  }

skip (y/n): n
  EXIT PRO-GOAL  append
X 1 <-T 0 ::= BOTTOM
X 2 <-T 0 ::= BOTTOM
X 3 <-T 0 ::= BOTTOM
{  }

  EXIT EXTG
X 1 <-T 0 ::= BOTTOM
X 2 <-T 0 ::= BOTTOM
X 3 <-T 0 ::= BOTTOM
X 4 <-T 0 ::= BOTTOM
X 5 <-T 0 ::= BOTTOM
X 6 <-T 0 ::= BOTTOM
{  }

  EXIT RESTRC
X 1 <-T 0 ::= BOTTOM
X 2 <-T 0 ::= BOTTOM
X 3 <-T 0 ::= BOTTOM
{  }
```

```
  EXIT LUB
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 2

ADJUST

TRY CLAUSE 1
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  CALL UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT LUB
```

```
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 1

TRY CLAUSE 2
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  CALL UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(INT,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= .(INT,OR(T0,[]))
X 4 <-T 0 ::= INT
```

```
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X3/(2)},{X6/(0)}  }

  CALL PRO-GOAL  append
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  }

skip (y/n): n
  EXIT PRO-GOAL  append

X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT EXTG
X 1 <-T 0 ::= .(INT,[])
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= []
X 6 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X2/(0)},{X3/(2)},{X6/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= .(INT,[])
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
{  {X1/(1)},{X3/(1)}  {X2/(0)},{X3/(2)}  }

  EXIT LUB
X 1 <-T 0 ::= OR(.(INT,[]),[])
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR(T1,[]))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 2

ADJUST

TRY CLAUSE 1
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }
```

```
  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  CALL UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT LUB
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 1

TRY CLAUSE 2
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }
```

```
  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  CALL UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(INT,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= .(INT,OR(T0,[]))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X3/(2)},{X6/(0)}  }

  CALL PRO-GOAL  append
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  }

skip (y/n): n
  EXIT PRO-GOAL  append
X 1 <-T 0 ::= OR([],.(INT,[]))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }
```

```
  EXIT EXTG
X 1 <-T 0 ::= .(INT,OR([],T0))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= OR([],.(INT,[]))
X 6 <-T 0 ::= OR(.(INT,[]),[])
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X2/(0)},{X3/(2)},{X6/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= .(INT,OR([],T0))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
{  {X1/(1)},{X3/(1)}  {X2/(0)},{X3/(2)}  }

  EXIT LUB
X 1 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR(T1,[]))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 2

ADJUST

TRY CLAUSE 1
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }

  CALL UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
{  }
```

```
  EXIT UNIF-VAR
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT LUB
X 1 <-T 0 ::= []
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

EXIT CLAUSE 1

TRY CLAUSE 2
  EXIT EXTC
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  CALL UNIF-FUN
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }
```

```
  CALL UNIF-FUN
X 1 <-T 0 ::= .(MAX,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],.(INT,T0))
X 4 <-T 0 ::= MAX
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= MAX
{  {X1/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  }

  EXIT UNIF-FUN
X 1 <-T 0 ::= .(INT,MAX)
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= .(INT,OR(T0,[]))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= MAX
X 6 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X3/(2)},{X6/(0)}  }

  CALL PRO-GOAL  append
X 1 <-T 0 ::= MAX
X 2 <-T 0 ::= MAX
X 3 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
{  }

skip (y/n): n
  EXIT PRO-GOAL  append

X 1 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))
{  {X2/(0)},{X3/(0)}  }

  EXIT EXTG
X 1 <-T 0 ::= .(INT,OR([],T0))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
X 4 <-T 0 ::= INT
X 5 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))
X 6 <-T 0 ::= OR(.(INT,[]),[])
{  {X1/(1)},{X3/(1)},{X4/(0)}  {X1/(2)},{X5/(0)}  {X2/(0)},{X3/(2)},{X6/(0)}  }

  EXIT RESTRC
X 1 <-T 0 ::= .(INT,OR([],T0))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= .(INT,OR([],T0))
{  {X1/(1)},{X3/(1)}  {X2/(0)},{X3/(2)}  }

  EXIT LUB
X 1 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))
```

X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR(T1,[]))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))

EXIT CLAUSE 2

ADJUST
Abstract interpretation is finished

The result is: append
X 1 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR(T1,[]))
X 2 <-T 0 ::= OR([],T1) T 1 ::= .(INT,OR([],T1))
X 3 <-T 0 ::= OR(T1,[]) T 1 ::= .(INT,OR([],T1))

# procedure used in program:	1
# clauses used in program:	2
# program points used in program:	7
# Goals used in program:	1
size of the static call graph used in program:	1
# tail-recursive procedure in program:	0
# locally-recursive procedure in program:	0
# Mutually-Recursive Procedures in Program:	0
# non-recursive provedures in program:	1
time of computing :	5.51 sec
# iterations (solve_goal) :	4
cpt1 =	8
# recursive pairs:	0
# substitution-preserving recursive pairs:	0
# substitution-increasing recursive pairs:	0
# substitution-decreasing recursive pairs:	0
# non-comparable recursive pairs:	0
# abstract domains:	1
# elements explored in abstract domains:	1
# elements stable in abstract domains:	1
# elements in foundation:	0
maximal length of the lattices:	2
maximal size of the lattices:	1
average length of the lattice:	2.00
average size of the lattice:	1.00
# updates in lattices:	3
average number of updates:	3.00
maximum number of updates:	3

dynamic size of lattices: 0
average dynamic size: 0.00
maximum dynamic sizes: 0

clauses actually visited: 0
clauses saved and not visited: 8

Do you want to see the elements not in foundation (y/n) ? n
Do you want to see the partial orderings (y/n) ? n